Tampere University

Mohammadali Akbarisamani

SERVICE BASED ARCHITECTURE WITH SERVICE MESH PLATFORM IN THE CONTEXT OF 5G CORE

Information Technology
Master Thesis
November 2019

ABSTRACT

Mohammadali Akbarisamani: SERVICE BASED ARCHITECTURE WITH SERVICE MESH PLATFORM IN THE CONTEXT OF 5G CORE
Master Thesis
Tampere University
Information Technology
November 2019

A gradual transition from virtual machine based design to new architectures such as service mesh is happening. The service mesh engages with remodeling of application architecture, which ultimately led to a shift in where the service is executed not a shift in what. In this model which is the service mesh, request traffic has an important role in deciding how to remodel the architecture of network.

The objective of this thesis work based on a study item for 3GPP release 16 is Enhanced-SBA. One of the key issues is related to HTTP/2 messaging, which doesn't have an inbuilt load balancing, overload control or failure recovery mechanisms. One potential solution is in addition to using new radio, consider enhancing with the cloud native grown principles like a service-mesh.

As for the used methods in this thesis, using a service mesh platform being our proposed solution for SBA, research and studies were made between different technologies and Istio was chosen as the most suitable framework for our purpose. It is an open source service mesh platform designed by Google, IBM and Lyft. It layers transparently onto existing distributed applications which lets you run a distributed microservice architecture and runs primarily on Kubernetes.

Istio uses sidecar proxies called Envoys to reduce the complexity of managing microservice deployments by providing a uniform way to secure, connect, and monitor microservices.

The results of this work are presented with conducting an extensive study, which was made on request routing and load balancing principles. Also, applicability to multiple instances of a micro-services (Network Functions), based on different conditions like load, different headers, etc.

The extra latencies introduced by each Envoy was measured under load and different factors affecting the results. However, it must be considered that the measured results depend on the environment where the framework is running, the features enabled, etc. (less than 1 ms in this case).

Keywords: Microservices, 5GC, Service mesh, containers.

PREFACE

This thesis has been done to conclude my Master's degree studies in Information Technology with Communication Systems and Network major. The thesis has been in collaboration with Nokia Bell Labs Standardization Research Lab, Espoo.

I would like to express my thanks to my supervisor at university Assistant Professor Davide Taibi for his support.

Special and sincere thanks to my team in Bell Labs, Kaisu Iisakkila for giving me the opportunity to work with the team of experts. My instructor, Tero Lötjönen, as well Pekka Korja and Markus Isomäki to work alongside them and for their excellent guidance and support, help and comments during the work.

I want to thank my friend Amirhossein Moshfeghifar and all my friends who supported me during this work.

Most importantly I would like to thank my family, especially my father, without them I wouldn't be where I am today. I would like to dedicate this thesis to my family for all their effort, love and support every step of the way.

Tampere, 3 November 2019

Mohammadali Akbarisamani

CONTENTS

1. INTRODUCTION 6
   - 1.1 Concept of Service Mesh 8
   - 1.2 Problem Statement 9
   - 1.3 Thesis Structure 11

2. RESEARCH LITERATURE AND RESEARCH BACKGROUND 12
   - 2.1 Expectations for SBA 12
   - 2.2 On the Cloud Native 5G Architecture 13
   - 2.3 Technology Evolution and Transition To SBA 14
   - 2.4 5G Core Components 16
   - 2.5 About SBA in 5G Core 19
   - 2.6 Service Mesh and Microservices 22
     - 2.6.1 Service Mesh Platforms 24
   - 2.7 Tools and Technologies 25
     - 2.7.1 Kubernetes and Main Components 26
     - 2.7.2 Operation and Monitoring Tools 28

3. RESEARCH METHODOLOGY 31
   - 3.1 Service Mesh Technologies 31
   - 3.2 Proposed Solution 33
   - 3.3 Istio 33
     - 3.3.1 Traffic Management Between Proxies 34
     - 3.3.2 Core Components 35
   - 3.4 Technical Implementation of Project 42

4. EXPERIMENTS AND RESULTS 43
   - 4.1 Testing in Different Environments 43
   - 4.2 Actual Setup and Test Application 47
   - 4.3 Latency Measurements 52
   - 4.4 Performance Analysis 56
   - 4.5 Istio with Mutual TLS 58
   - 4.6 The 5G Use Case with Istio 59
     - 4.6.1 5G Core Functions as Microservices 60
     - 4.6.2 Connectivity Verification 62
   - 4.7 Multicluster Istio 63

5. CONCLUSION 65

REFERENCES 67

LIST OF FIGURES

Figure 1. *Transition to microservice architecture. 8*
Figure 2. *Migration to cloud native microservices. 11*
Figure 3. *Key areas of a typical 5G system. [12] 17*
Figure 4. *Based on 5G System architecture - SBA - 3GPP TS 23.501 V15.2.0 18*
Figure 5. *Definition of service based architecture. [23] 20*
Figure 6. *: Elasticsearch, Logstash, and Kibana, approximate scheme. 29*
Figure 7. *features of four service mesh options for our solution. [47] 31*
Figure 8. *Istio Architecture with Data plane and control plane virtual interaction. (From Istio [4]) 35*
Figure 9. *Pilot architecture. (From Istio [4]) 37*
Figure 10. *Istio traffic routing and management, content or traffic wise. (From Istio [4]) 40*
Figure 11. *Request flow for Ingress and Egress trafficof a service. (From Istio [4]) 41*
Figure 12. *Kubernetes Dashboard UI. 46*
Figure 13. *Bookinfo architecture without its sidecars. (From Istio [4]) 48*
Figure 14. *Bookinfo architecture with sidecars. (From Istio [4]) 49*
Figure 15. *Network graphs generated by Kiali –Bookinfo and Simple-webserver. 51*
Figure 16. *Network graphs generated by Kiali showing ingress. 52*
Figure 17. *Istio Observation tracing tools (Jaeger). 53*
Figure 18. *Jaeger detail request tracing. 54*
Figure 19. *Nginx application architecture deployed with Istio. 55*
Figure 20. *Fortio generated network load graph comparing with or without sidecars. 58*
Figure 21. *NRF connectivity and adaptability. 61*
Figure 22. *Connection between NRF and other services. 62*
Figure 23. *Network graph generated by Kiali. 63*
Figure 24. *Single control plane multi cluster based on Istio documentations.[4] 64*
Figure 1. *Sending requests from Ingress to “Nginx proxy 2”. 70*
Figure 2. *Sending requests from Ingress to ”Nginx proxy 1”. 70*
Figure 3. *Sending requests from Ingress to “Nginx-v1” (1 hop). 71*
Figure 4. *Sending requests from Ingress to “Nginx proxy 2” with TLS enabled. 71*
Figure 5. *Sending requests from Ingress to “Nginx proxy 1” with TLS enabled. 72*
Figure 6. *Sending requests from Ingress to “Nginx-v1” (1 hop) with TLS enabled. 72*
Figure 7. *Sending requests to “Nginx proxy 2” without Istio (only Kubernetes). 73*
Figure 8. *Sending requests to “Nginx proxy 1” without Istio (only Kubernetes). 73*
Figure 9. *Sending requests to “Nginx-v1” without Istio (only Kubernetes). 74*

LIST OF SYMBOLS AND ABBREVIATIONS

3GPP	The 3rd Generation Partnership Project
5G-AN	5G Access Network
5GC	5G Core Network
AF	Application Function
AMF	Access and Mobility management Function
API	Application Program Interface
AUSF	Authentication Server Function
CD	Continuous Delivery
CI	Continuous Integration
DN	Data Network
DRA	Diameter Routing Agent
DSC	Diameter Signaling Controller
E2E	End to End
EPC	Evolved Packet Core
eSBA	Enhanced SBA
K8s	Kubernetes
KPI	Key Performance Indicator
NAS	Non-Access Stratum
NE	Network Elements
NEF	Network Exposure Function
NF	Network Function
NG	Next-Generation
NRF	NF Repository Function
PCF	Policy Control Function
QPS	Query Per Second
RAN	Radio Access Network
Rel	Release
SBA	Service Based Architecture
SBI	Service-Based Interface
SMF	Session Management Function
SOA	Service Oriented Architecture
TLS	Transport Layer Security
UDM	Unified Data Management
UDSF	Unstructured Data Storage Network Function
UE	User Equipment
UPF	User Plane Function
URL	Uniform Resource Locator
VM	Virtual Machine
VNF	Virtual Network Functions

1. INTRODUCTION

Mobile core networks have been evolving to cloud and will continue to evolve towards the cloud native principles. Drivers for this are the need for flexible support a wide variety of business models, demands of new high-performance services (like low latency and high availability service for the industry and enterprise sectors), efficient dealing with unpredictable demand, managing the growing network complexity, enabling rapid introduction of the innovation to market. By deploying core networks using the web-scale proofed cloud native technologies would and components as ground enable taking networks to the next level and create increased productivity, service agility and diversity. [1]

In the telecom industry, this means, among other things, that applications are implemented as stateless modular, independent microservices and easily deployed on several infrastructures. [2] Expanding the complex network software functionalities, developers would have problems with monolithic architecture. Monolithic architectures cannot be scaled, upgraded or maintained simply to satisfy CI (Continuous Integration)/CD (Continuous Delivery) principles. The proposed solution is a new software architecture, where the complex tasks are divided into independent subtasks exchanging information. Generally speaking, cloud native applications are typically built as a set of microservices, that run in containers.

This development also includes the trend to unify the common functionalities from each individual application. These kinds of functionalities can be load balancing, failover support, health monitoring, version trialing, service discovery and selection as well as authentication and encryption. Because these functionalities are application independent and can be utilized widely, they are available as open source SW. HTTP is typically used as messaging protocol between the microservices, even there are other existing communication protocols.

Service mesh is an emerging trend for HTTP microservice communication, which allows outsourcing common functionality to a ”sidecar proxy”. This is the starting point of breaking service into microservices and the concept of service mesh. With the advent of increasing interest in using service mesh, many web-scale social media platforms deployed different types of service mesh in their architecture. When it comes to considering

a network with service mesh deployed inside, the main point is that the architecture is totally dependent on this new topology.

In general point of view, service mesh is a specific infrastructure layer enabling communication between services in an organization, which considers new architectures appeasing requirements of a new environment for the application. In fact, service mesh contains an array of network proxies alongside with the corresponding application code, and what's interesting is that the application doesn't need to be aware.

In the cloud native model, Kubernetes (a system for orchestrating containers) handles the internal system in which several thousands of instances, compose a service. Logically, an application is the sum of services created based on the served requests. The service mesh is considered as a separate automated connectivity layer, so here the question is where the service mesh came from. By taking a glance at some similar concepts in this area, we easily find that service mesh does not provide new and specific features, but mostly presents a new point of view in the context of network architecture

so that solve some basic problems such as optimized management of running time of applications.[3]

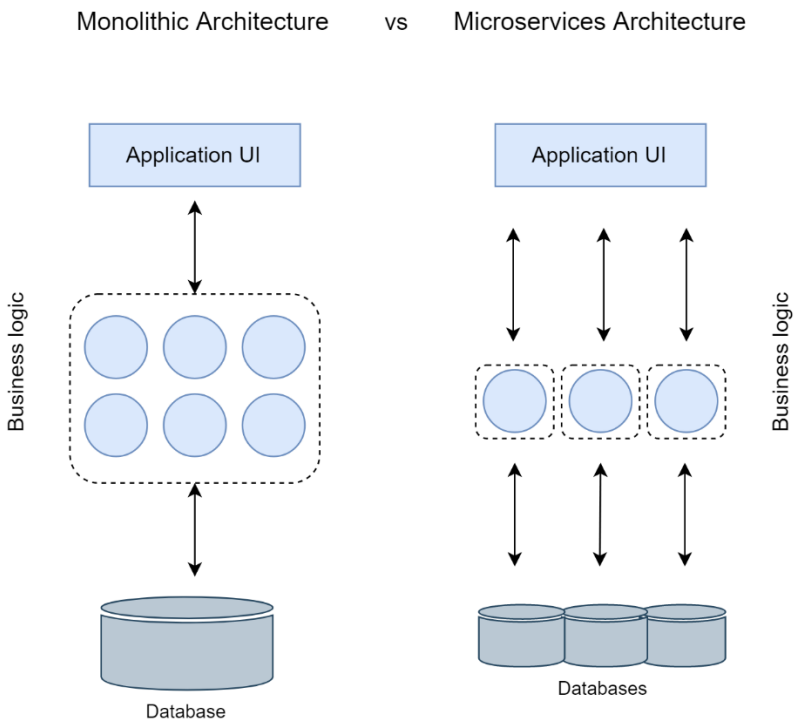

Figure 1. *Transition to microservice architecture.*

1.1 Concept of Service Mesh

Previously, service providers were running their services on some virtual machine (VM). VMs, are designed to perform as a real computer in a virtual space. They could be implemented using certain hardware, software, or a combination.[4]

In the recent decade, a gradual transition to new architectures such as service mesh is happening. The service mesh engages with the remodeling of application architecture, which ultimately led to a shift in where the service is executed not a shift in what. In this model, request traffic has an important role in deciding how to remodel the architecture of the network.

The former solution for optimized handling of request traffic, handled by a “web tier” which finally redirecting traffic to a reference part in the form of database libraries. Using database libraries, the app layer communicates properly with the backing libraries for managing services required by requests in an optimized way. It was looking good until

the recent decade, but this was not the model to undergo heavy load in some layers and urged the service owners to seek for a new configuration.

Later the concept of microservice was found. In fact, when some huge web-scale companies, both in terms of infra and service, like Google, Facebook, Netflix, Twitter found the monolith is no longer the responding and scaling as they expected, the microservices therefore as an alternative was offered. In fact, microservice architecture provides the possibility of data transfer between smaller, loosely coupled and independently deployable services called microservices.

Having this circumstance, the microservices were introduced, almost at the same time that east-west traffic meaning data packets from server to server within a data center was also introduced. To handle the heavy load of request traffic there should be almost the same runtime for services, it is executed by “fat client” libraries. These libraries arrange the request delivery and manage the corresponding services and let the developers develop the network or make any changes. In details, these libraries offer functionalities such as load balancing, routing, circuit breaking, and telemetry. Finally, the very first service mesh was formed by these libraries to run the services of applications, in the same way, disregarding the differences between.[3]

1.2 Problem Statement

There is going to be a bigger signaling load difference between 5G and each of the previous technology generations, therefore in this sense, strong strategic planning for designing and proposing new architectures is needed. One of the most important things in designing new architecture is that it should be able to handle very heavy service load, therefore the main focus is almost on proposing architectures good at management of services.

Service Based Architecture (SBA), or in another term, Service Oriented Architecture (SOA) is a new strategy standing beside other data transfer management solutions like messaging-oriented middleware or API Gateways, in which the main emphasis is on designing architectural approach based on the service. In practice, it is a new topology in which very high scale architecture breaks. Consequently, as mentioned before, it is ultimately a shift in where the functionality is located. Therefore, as one of the best proposals for 5G technology implementation, SBA is going to be investigated here, with an emphasis on service mesh platform. But it is important to mention at the beginning of this investigation, that SBA, with service mesh platform, is only as one of proposals and

not the one totally evaluated or confirmed as the best partial solution for 5G implementation.

To better understand the necessity of service-based architecture and in particular service mesh platform for managing the complexity of service communication, let's consider a typical model of handling web application in around 10 years ago. In this model, we had separate layers each pair of layers could communicate mutually. Since there were only two hops, there was only communication logic between hops, allowing a limited scope of communication under the code of each layer. While working for a fairly long time, gradually this architectural approach was urged to change.

The web-scale players faced with heavy traffic and requirements, therefore switched to another seemingly better approach, a cloud native approach: in which the application layer is composed of microservices. In this situation, the most complex challenge in realizing microservice architecture is not building the services themselves, but the communication between services. These systems handle a generalized communication layer in the form of a "fat client" library to respond to the traffic. Along with this change the service mesh would be implemented as outer infrastructure to better support independent microservices.

This provides an ample opportunity for independency and is realized valuable, but its value goes further even to the management of all inbound and outbound traffic.[5]

Before the introduction of this new architectural approach, those mentioned companies had forms of new infrastructure for managing services communication, and some of them found some other use outside of their origin companies, in other words, initially being sensitive to the details of their surrounding environment and allocating dedicated system for the task, later hit some problems, and ultimately led to the thinking of a shift in where the service is located.[6]

Service mesh architecture is not a basic requirement for the 3GPP specified SBA, but is an advanced and also a bit completing deployment option for the defined 5GC communication procedures. Service mesh platform within a service based architecture is going to be evaluated in this work, to affirm its capacity and productivity in responding to requirements for 5G technology implementation.

The aim for this thesis is to experiment and understand the practicality and feasibility of using a service mesh framework for SBA in 5GC and to get an understanding of both advantage and possible disadvantages of this deployment.

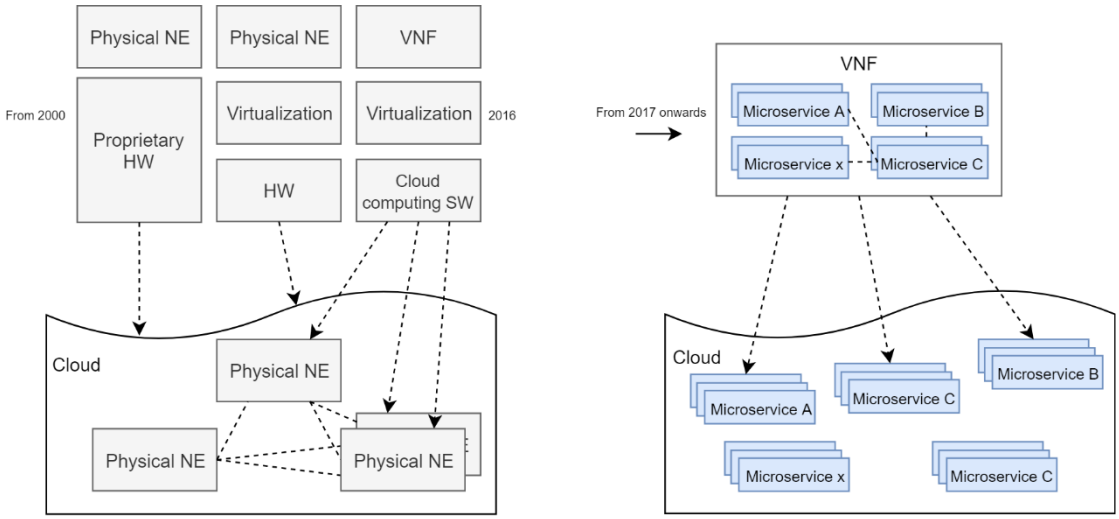

Figure 2. *Migration to cloud native microservices.*

1.3 Thesis Structure

This thesis is arranged as follow: in the next chapter we have research literature and background review, then we have a research methodology in the third chapter, that investigate the method of the study of this thesis and also, talk about the way by which measure the key factors. Later the experimentation and the results will be presented and finally, the conclusion is provided by the last chapter.

2. RESEARCH LITERATURE AND RESEARCH BACKGROUND

In this thesis, we are aimed at investigating the service based architecture and in particular the concept of service mesh, in a 5G network. In this network we expect to present diversified service requirements using cloud native 5G Core network architecture where SBA is initially specified in Rel. 15 and will be further enhanced in Rel. 16. In the following sections we review the expectations for SBA, related cloud native architecture and effective measures against the target of study, 5G Core (5GC) components, service mesh and the technologies used. [7]

2.1 Expectations for SBA

This section is aimed at briefly scrutinizing the major feature of this new architectural approach in the context of a 5G core and discussing the expectation against the former approaches regarding some key point including message management solution.

Firstly let's take into account features claimed to be certain features of the new architecture, in this regard any service mesh has specific attributes regarding the table stakes, including: robust load balancing algorithms, cascading failure prevention (circuit breaking), resiliency features, control over request routing, the advantage of introducing and managing Transport Layer Security (TLS) termination between communication endpoints, set of measurements to provide instrumentation at the service-to-service layer.

Each of these features is considered among the features of message management solutions. Since the concept of SBA is tied to the concept of microservice, so we need to have a better sense of microservice in telecom industry. In order for better understanding the concept of microservices, we need to have a look at the related changes in the last few decades. Since the advent of the Internet the software industry is experiencing rapid changes on a large scale which are usually described with words, such as "cloud computing," "web-scale" and "software defined networks". Ultimately this rapid growth is pushing applications to a maturity level typically described as "cloud native." Therefore, in the telecom industry, this means stateless microservices with a high degree of modularity.

In this situation, almost all applications are composed of easily managed and deployed services across a multitude of infrastructures. Thus, in order to maintain a competitive

edge in telecom industry, the principles of a microservices-based architecture are essential in meeting a cloud native maturity level. While necessary to deliver on the promise of assuring continuous innovation and software deployments, however it may bring fundamental shift in where services are located. Now let's evaluate the new solution in a 5G core control plane. [8]

For the 5G service-based architecture, service meshes are the new solution. For instance in order to solve problems, including scaling, interworking, and fault isolation in a web-scale environment, service meshes is used to address the challenges related to the communications layer between 5G network functions such as how to automate, secure, scale, simplify, as well as optimize the services which are deployed across a multi-cloud infrastructure distributed over the core, region, and the edge of the network to provide users and applications with the connectivity they need.[8]

2.2 On the Cloud Native 5G Architecture

In the 5G era, presenting diversified service requirements is a key issue. In order to enable diversified service requirements, it is necessary to have a cloud native 5G architecture design and implementation. As a result, to provide costumers and any individual with a real-time, on demand social experience, it requires a digital transformation, an end to end (E2E) coordinated architecture, possessing features such like being agile and automatic.

To make the digital transformation to happen, comprehensive cloud adaptation of networks, operation systems, and services is a necessity. This strategy toward 'all cloud' environment is an exploration into software architecture, hardware resource pools, and automatic system deployment. Mainly this transformation focused on transforming networks using a network architecture in which all service applications are running on the cloud data center, or in other words a cloud native architecture.

By separating control and user plane, 5G core networks will make simple the whole signaling, which is achieved using component-based control planes, unified database, and programmable user planes, after all, it allows for the deployment of distributed gateways. In a 5G network, operators need to flexible additional network slices to better serve costumers needs in this line they may change the functions into customized network functions. 5G Core control plane has adopted some web-based principles and protocols like HTTP/2 communication in order to benefit from the rapid development of the web-scale signaling and event processing capabilities. SBA is defined in the 3GPP Rel 15 5GC architecture, but realization and its feasibility with service mesh principle are to be seen.

As already mentioned, a single cloud native 5G network infrastructure can meet diversified service requirements. Cloud-native end-to-end network architecture has the following features:

- To meet diversified service requirements and provides data center-based cloud architecture and support various application scenarios, a logically independent designed network slicing on single network infrastructure is implemented.
- All services and the corresponding architecture are based on logically independent network slicing.
- Reconstructing radio access networks (RAN) by cloud radio access network utilization, so that support all standards and connections of 5G network.
- Implementation of temporal network functions configuration with the use of simplifying core network architecture.
- Reducing operating expenses through agile network operation and management (O&M) using implementation of automatic network slicing service generation, maintenance, and termination.[9]

2.3 Technology Evolution and Transition To SBA

When the network functions are changed from network element (NE)-centric perspective to a modern cloud native architecture to individual independent elements and form a cloud native microservice, the services can be dynamically deployed throughout the network to meet the scale, KPI, and SLA requirements of an end-to-end service. There are some challenges that urged the providers to seek for a new solution, here the service mesh. [2] For instance, due to some reasons such as incompatible interactions and varying degradations, as well as unpredictable failure modes the complexity of a distributed service-based architecture is further increased with microservices.

These challenges are the primary reason why a service mesh is needed. In order to access to a secure, reliable, and resilient communications layer, handling the delivery of signaling requests through the complex, dynamic, and ever-changing topology of services, a service mesh is the unique solution at the time, moreover it comprises a modern, cloud native 5GC deployment to meet our expectations for end-to-end communication. Here to measure the expectations, one can say that building large-scale, distributed signaling systems to provide service-based and web-scale solutions critical to the success

of multi-cloud and multi-vendor 5G networks. Along with this, automating, securing, scaling, optimizing, and simplifying the signaling communication for 5G services across a multitude of less reliable infrastructures is a significant challenge.[9]

Altogether, this project is aimed at inspecting these measures and contrasting against the real expectations of service mesh in a 5G core. In detail, we can say that the two main components comprise a service mesh: A service mesh is split between a signaling plane of proxies and a mesh control plane in which the signaling plane is made of service proxies. All 5G core signaling traverses a signaling plane composed of lightweight service proxies. The second component, centralized mesh control plane, does coordination and application of management, operations, and policies to all service proxies

Considering a 5G core signaling plane, the service level communication is categorized as east-west. The first role of a service mesh is the management of service communications for east-west traffic flows, which refers to the transfer of packets between services (or servers) within an administrative domain. In other words, a service mesh typically controls and monitors the internal traffic of microservices. Its data plane implemented alongside application code, and its control plane interacts with these proxies.

The transformation from a NE-centric view to a modern cloud native architecture needs service meshes. Popular open-source examples of service mesh are Istio-Envoy and Linkerd (Conduit).

Diving into the history of the service mesh, one could find that, for many companies using Docker and Kubernetes, fortunately they have solved the problems of deployment. But the problem is that they do not guarantee any solved runtime. Here is where the service mesh urged to be invented. Having the “solving deploys” in the mind, let’s consider Using Docker and Kubernetes, what in practice is observed is that these things decrease the increasing operational burden to deployment which eventually results in a dramatic reduction in the expenses of taking in microservices.

To address the reason for this huge step forward, we can say that Docker and Kubernetes offer powerful virtual tools at all the right levels alongside with standardizing the patterns for packaging and deployment. But as already mentioned, there is no guarantee for any solved runtime, after the deployment, the app still has to run. Thus, in order to equally standardize the runtime operations of all applications, we need to change the strategy, here turning to the service mesh.

This solution controls and measures traffic between apps or services in datacenter parlance, in a uniform, global operation. Thus, by adopting microservices, the only tool for standardizing the application’s runtime is standardizing the management of the traffic of

both incoming requests and issuing outgoing requests. In details, for analyzing and then operating on this traffic, service mesh provides ways to ensure reliability, security, and visibility by running a standardized mechanism for runtime operations. After all about service mesh, now it's time to go for microservice. [7]–[9]

In 2005 Dr. Peter Rodgers for the first time invented the phrase "Micro-Web-Services" and used it at his paper accepted by a cloud computing conference. Later in the spring of 2011, the term "microservices" invented at a conference of software architects. Gradually after that, many companies such as web-scale players switched to this approach because of gained popularity due to supporting many of the changes in areas, such as server utilization, multi-core servers, mobile devices, web apps, cheap RAM and etc. Also, other architectures such as Service-Oriented Architecture (SOA) have been developed that have close functionality to microservices functionality. [9], [10]

Microservices can be the answer to the problem of being unable to scale monolithic architecture, as they break down complex tasks into smaller processes that work independently of each other. An extremely important thing about beginning to scale microservices is to define target architecture before any scaling, so that avoid any chaos and worse properties in the IT landscape. [10], [11]

2.4 5G Core Components

State of the art technology is Next-Generation Standalone core in 5G mobile broadband technology defined in the 3GPP release 15. Tracing back the principle of 5G, one can find 5G network principles and system, as an evolution of Evolved Packet Core (EPC), but the main difference is in that in the new generation, we want to use the service based architecture as the basis. 5G architecture is designed based on how service and network function can interact mutually by service based architecture, which uses service-based interfaces and can be better understood by referring to reference point which also uses service-based interfaces.

In reference point, the interaction between network function services is defined by connectivity between any two services such as AMF and SMF. In service based, network functions enable the other predefined network functions within the control plain so that can access to their services. Moreover, there is another interesting thing with the 5G mobile broadband, and that is, it guarantees the possibility for diverging architectures for extension of new services. Finally, within the 5G system, architecture features a combination of automation and programmability. [7]

The 5G system will have the following components as the fundamental components of the system: 5G Access Network (5G-AN), 5G Core Network (5GC) and User Equipment (UE). However, in this subsection, only the core components of 5G mobile broadband are investigated and some key features and points will be discussed. It is notable to mention that there are multiple 5G radio (NR) deployment options together with EPC core (Non Standalone, NSA), but in order to support properly a wide range of services, 5G requires a specific system architecture core network, which is often referred to as Next-Generation Standalone core (NG (SA) core).

In general, while the most concentration of improving the architecture of NG core is to present a network fully supporting services as enhanced mobile broadband (eMBB), but in near future, for sure there will be new emerged service dimensions. However, the core component of 5G network must be designed and implemented in a way such that provide as many functionalities as is or will be needed.

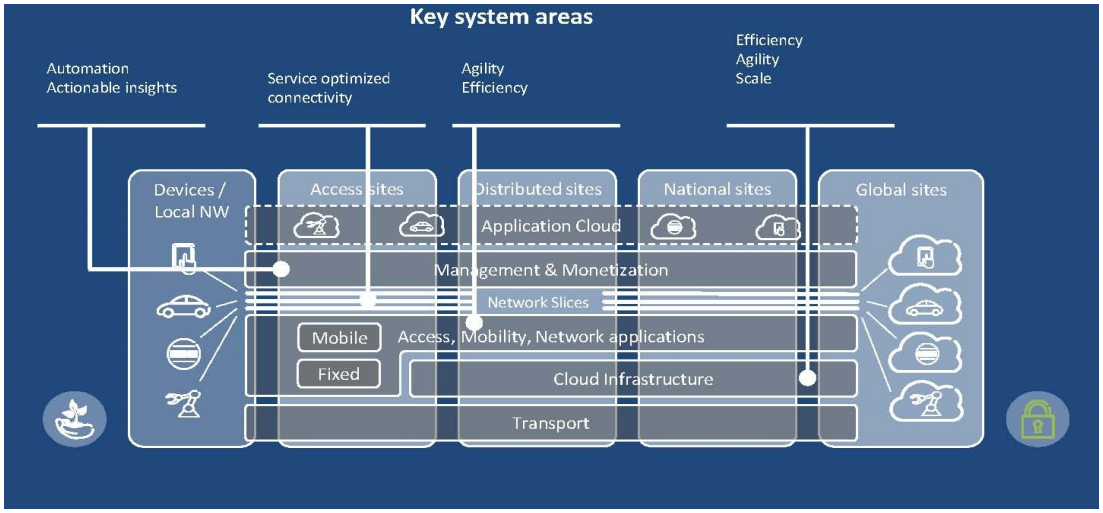

Figure 3. *Key areas of a typical 5G system.* [12]

These component are namely: Authentication Server Function (AUSF), Access and Mobility Management Function (AMF), Data network (DN), Policy Control function (PCF), NF Repository Function (NRF), Network Exposure Function (NEF), Unstructured Data Storage network function (UDSF), Policy Control function (PCF), Session Management Function (SMF), Unified Data Management (UDM), User plane Function (UPF), Application Function (AF) and finally (Radio) Access Network ((R)AN). [13], [14]

Toward supporting NG core requirements in our network, each of the functions must support its own functionalities, but this is not possible unless the communication between the functions and the corresponding authorities are best designed. While by looking at these elements, we maybe find some totally new functions, but also it may make us think

about the resemblance between some of the elements of this generation and 4G. Since any network always must support some basic functions of networking, some of the elements never will be removed, just getting enhanced to the state-of-the-art technology.

Some of the components such as AMF and SMF are essential in the 5GC functionality and can be used as examples here to understand the SBA principles while the others utilize the principles similarly, therefore just some of them are presented in the next few subsections.

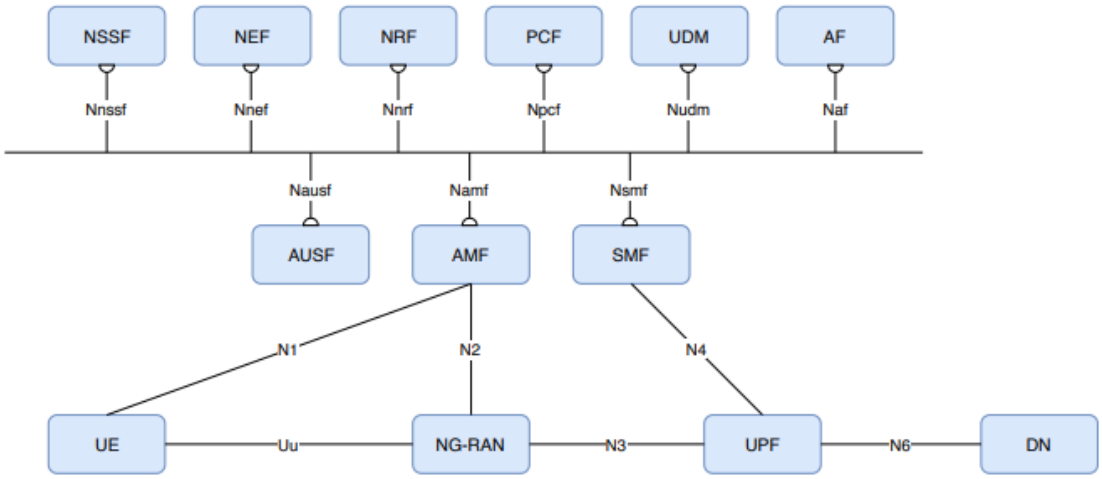

Figure 4. *Based on 5G System architecture - SBA - 3GPP TS 23.501 V15.2.0*

One of the vital components of a 5GC is Access & Mobility management function, which stands for managing access control and mobility, as a control plane component in the NG core network architecture of 5G. Within a 5G network, there is an important need for dynamic management for access and at the same time mobility according to network service based architecture. AMF as well comprises network slice selection which is very important in communication between the functions.

Regarding this communication, key tasks of AMF include: proper management of registration, Termination of Non-access stratum (NAS) signaling which is a functional layer managing the establishment of communication sessions and non-stop communications between core network and the user equipment, protection of NAS ciphering & integrity, Mobility Management, access management, managing Connection between points, managing the individual point reachability and security context management and authorization. As we see in the Figure 4, 5G control plane (the upper part) has a “bus” and service-based interface, which makes the architecture, service based, each network function may allow the others to access to its services. [15]

Session Management Function (SMF) as can be inferred by the name, performs session establishment, modification, release, there is a component in 5G which is called SMF. Moreover, IP address allocation & management for user equipment (UE), DHCP functions, termination of Non-access stratum (NAS) signaling relevant to session management, DL data notification, user plane function (UPF) traffic managing configuration for proper traffic routing are other relevant tasks of session management function. In details, in order to manage the user equipment, as one of three main components of 5G systems, we need to allocate an IP address, this is done according to some specific protocols by session management. [16], [17]

NRF is another core function which is important for this project as well, it is used to register all functions and capabilities in service based architecture, where other functions can request their communication peers. In a live network dealing with high flow of information, it is required to have some tool so that maintains functionalities related to network status, for instances in 5G network, which network functions (NFs) interact properly to respond and manage the request traffic, need to be maintained, moreover there should be some tool in order to discover services within the system, here is where the concept of network repository function makes sense.

Within a service-based architecture, we have the capability to allow the services to automatically register themselves and immediately configure themselves, which is very important in speeding up the network response. Having a glance at the following figure, we can see that this service-based architecture equips the network with the bus-type interface, therefore any node in the upper plane (control plane) has the ability to access the services of the other nodes if it is authorized to do so. One critical thing related to the architecture with the direct access to the service of each node is that it must have a mechanism to support discovery of service and maintaining the NF profile, NRF is implemented to do so and enable the architecture to accept the automatic registration and configuration of independent services. [17]–[19]

2.5 About SBA in 5G Core

Compared with 4G and 5G NSA, 5G core has a totally different control plane architecture which is designed and implemented based on the architecture for direct service to service communication. This type of architecture gives the ability to the control plane of our network to handle this data transfer. Service based architecture along with slicing, provides 5G system increased flexibility. Looking at Figure 5, we can see that service-based architecture possesses a service-based interface (SBI) which equips each of control

plane function with the proper communication with other functions, for instance, network repository function is authorized to act as a tool for registry of other functions.[20]

Service based architecture proposes a wide range of functionalities and accessories, from direct faster inter-communication between the functions and corresponding services to high level of security in several aspects.

Compared to the point to point architecture, operators prefer to choose SBA, because it is more compatible with the new networking models especially cloud native networking model. Critical functions to develop delivering diversified services include security, session management, mobility, and some other functions. [21]

The key point about 5G is that only service based architecture can support new trends in this technology such as cloud native infrastructure. After all, service agility is another very important requirement which allows the 5G network to provide many other options, and this is not possible unless adopting SBA. Therefore, a huge need to transition to SBA is felt. As illustrated in Figure.5, Briefly engaging with the practical implementation of SBA, one finds two necessary items for implementing the SBA are network function service and service based interfaces. [6], [22]

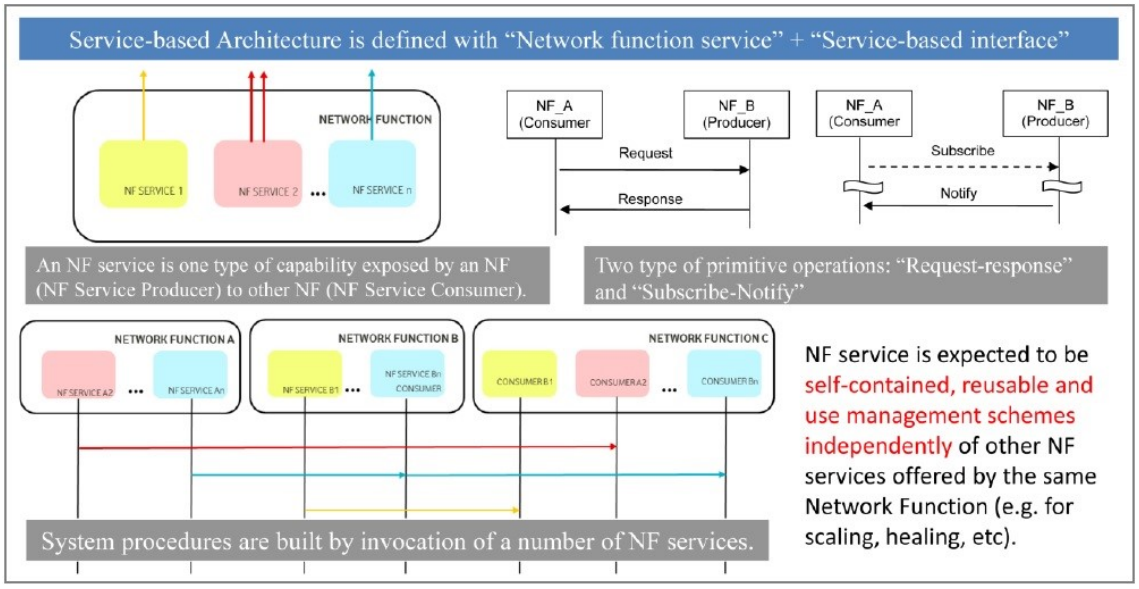

Figure 5. *Definition of service based architecture.* [23]

There are some benefits and targets to be achieved in using SBA in 5G Core, which is why there have been extensive study and research oriented toward service based design and implementation of the architecture. SBA provides a bus which is embedded in service-based interface, using this bus, each component of the 5G core would be authorized

to access to other one's services or any new services would automatically register themselves.

One of the most important targets of SBA is to enable 5G network to support scalability and fast diversified services which could be handled by a particular cloud native friendly architecture which would be the state of the art solution. [24]

To go over the status of SBA in 3GPP standards, in recent years, 3GPP is focused to expand the 5G system architecture to new 5G core, compatible with cloud native based infrastructure. In fact, SBA principles in 5GC are defined in Rel.15 and there is work ongoing under topic eSBA (Enhanced SBA) in rel. 16.

This is mainly advancing by the design of architectures depending on independent services and their communication. SBA therefore is affecting new and, in some senses, old 3GPP standards so that become as much compatible to cloud native as possible. [25]

Release 15 of 3GPP which is now finalized provides the technical specification of 5G regarding SBA. Mostly, it discusses architecture requirements, principles, and assumptions, and extends it to some key issues about SBA in 5G and later suggesting some solutions. It also defines the HTTP/2 as the messaging protocol between the SBA NFs.

Release 16 as a technical report under study, "studies and evaluates architecture enhancements on potential optimizations to the Release 15 Service-Based Architecture (SBA) in order to provide higher flexibility and better modularization of the 5G System for the easier definition of different network slices and to enable better re-use of the defined services." Also, studies on mechanisms to better support automation and high reliability of network function services. While not finalized, release 16, is going to suggest an enhancement to the service-based architecture in the form of 'Enhanced Service Based Architecture (eSBA)'.

Rel 16 will introduce Service Communication Proxy (SCP) that can have a role in service selection, load balancing, and other common functions. For example, a transparent vs outband vs proxy mode, weather to use sidecars are not, the interface and interactions with NRF, security setup method. [26]

It shall be noted that even the Rel 16 specified 5GC eSBA is not a pure micro service architecture, but rather the standardized messaging framework between the 5GC NF, which are still quite big monoliths by definition. Of course, the actual deployment of the NFs can evolve towards micro service architectures.

LTE core network (EPC) messaging principles are based on multiple protocol and Diameter is one of the major one. 3GPP has specified various methods for load balancing,

overload handling and failure recovery etc. situations. The issue with the LTE network was that with its fast expansion, along with data traffic unabated growth are leading to an unprecedented switch in the core network to Diameter signaling. It eventually led to the introduction of a new type of products called Diameter Signaling Controllers (DSCs), which efficiently uses routing signaling information to deal with the growth and expected congestion in signaling traffic. In EPC and IP Multimedia Subsystem (IMS) networks, most of the network elements interconnect using Diameter. In fact, in order to avoid network outage resulting from congestion in the signaling plane, in the case of more than one instance of any of the network element we can use this solution.

With the increasing use of LTE devices and applications, finally the operator faces with the exploded number of interfaces in the EPC. meaning that the EPC would have very high volume of Diameter messages. Therefore, to handle the situation, or in other words “to support scalability and reliability for Diameter in the EPC, a carrier-grade infrastructure is needed”. The Diameter Routing Agent (DRA) is introduced as the provider the carrier-grade infrastructure in the EPC. It helps support congestion control, centralizing traffic management, failover assurance, binding the session, vendor interoperability, offering new services by application selector function. [27], [28]

2.6 Service Mesh and Microservices

Considering the concept of structuring as a trend of breaking services in smaller services (and now microservices) in order to do load balancing, led to fundamental changes in the network architecture. Microservices are the product of redesigning of software architecture so that we have applications which are composed of small, independent parts, which work together through APIs. This allows developers to build systems in a way constructed with standardized units, because this microservice has a limited domain of working and takes action on a particular task independently. Therefore, as mentioned in previous sections, microservices are focused components designed to do only one limited independent task very well. [29]

Like mentioned before, when it comes to expanding the complex network functions with the scale, developers would have problems with the monolithic architecture, because monolithic architectures cannot be scaled, upgraded or maintained simply. The feasible solution from the web application communication is a new software architecture, where complex tasks are broken down into small subtasks, that communicate mutually. Monolithic applications have three main components: a database, user interface and an appli-

cation on the server, in which HTTP requests are processed, discussed with the database, and the reply sent to the browser. But in microservices architecture HTTP request/response are handled with APIs and messaging. [29]

Considering service mesh migration to cloud native, the infrastructure needs handling a new type of communication known as service-to-service communication, which is done by service mesh. Service mesh goes for secure delivery of requests, in particular those arising from cloud native application, through the network architecture. It is responsible for establishing and in some sense, managing the communication between independent services.

In TCP/IP, assuming that the underlying L3/L4 delivers data from point to point, one can consider the service mesh as a network modeling. However, since the network is not reliable, service mesh is responsible for the issue such as reliability and handling the faults. Also, one of key features of any network is visibility, therefore this network modeling, service mesh, stands for visibility and proper static handling of request traffic and corresponding application runtime via continuous monitoring, which is far better than what TCP presents. During recent decades, rapid growth of network scale and massive traffic requirements, have had demanded a new solution for architecture design and implementation. It moved toward introduction of the concept of microservices, which was actually resulted from splitting application layer to many services communicating independently. In other words, service mesh has been rooted from a shift in where functionality is placed. Replacing functionalities helped web applications better manage the complexity of service communication. [30]

Any service mesh should have the following attributes, to meet the requirements in the cloud native:

- Resiliency features which include, but not limited to: timeouts, deadlines, and retries.
- Preventing cascading failure or in other words, circuit breaking
- Any service mesh must have some algorithms vigorously balancing the load.
- Transport Layer Security, in fact, any service mesh should provide and manage TLS termination between communication endpoints.
- Proper and automated request routing management is another feature of importance.
- Delivering requirements at the service-to-service layer by divergent type of metrics. [30], [31]

2.6.1 Service Mesh Platforms

In this subsection, we try to introduce some well-known platforms of the service mesh, and briefly emphasize on specific features of them.

The first is **Linkerd** which is the first open source network proxy, which is implemented as service mesh, practically with the Linkerd, there is no need to change your code and mostly used for Kubernetes. however, it is also used for some other frameworks. One of its most important tasks is that all services within the network will be run easier with a high level of security, due to runtime visibility and real time debugging. Also, is help provide more secure and reliable communication between the services in large production system. Technically speaking, Linkerd provides a control layer of abstraction on top of communication between the services to manage and control them so that optimized the runtime. Cross-service communication is also considered as a vulnerable part of the system, which is continuously monitored by this service mesh. [31]

Linkerd2 (Conduit) is a new version of Linkerd, which is a rewrite of Linkerd in Golang and Rust languages for Kubernetes. Its most compelling option compared to other products, such as Istio is its simplicity. Linkerd2 is orders of magnitudes faster than Linkerd, moreover, it is really smaller in the code size. Unlike the older version, Linkerd2 uses Rust for data plane proxy, which is directly a great help to debugging many bugs and memory issues. It brings a tight default coupling between the services of upper and lower part of service mesh (data plane and control plane). However, it lacks some features, such as distributed tracing. [31]

Consul is another product presenting the service mesh requirement, with its specific features. Latest version of Consul is the featuring function connect for both data and control plane in the form of a single Go binary. The main advantage of Consul is that it allows you enable connect across services on Kubernetes and also join them to services on virtual machines outside, however it operates with non-centralized control plane services or converging architecture for control plane services. Since it is a clear separation between layer4 and layer7 with a simple enough design. [31], [32]

Istio is one of the pioneer services meshes in many new ideas, it is a stable, powerful and well-implemented product which is backed by huge companies such as IBM and Google. However, compared to other service meshes, Istio is more complex and modular, which is not of interest of developers, in other words it is not that kind of service mesh which could be implemented anywhere easily. Along with this disadvantage, some features such as automatic sidecar injection make it to an excellent service mesh. Istio, as

already mentioned, usually pairs with Envoy by default, and provides a control plane to handle a various service proxy. [3], [31]

2.7 Tools and Technologies

In the last six years, containers have been introduced and used to package applications in a way they can access a specific set of resources on a host's operating system, in other words in a microservice architecture, each service is packaged in a separate container. Containers have altered the way software organizations deal with application, such as building, shipping, and maintaining applications. Containers are scalable and also ephemeral, meaning they come based on needs. As the number of containers and also services increase, the management of the containers become more complex. [33]

Docker is first released in 2013 which is the most ubiquitous containers. It is a computer program to containerize within the system. Docker runs isolated containers carrying materials such as application tools, libraries, and other relevant files. The nice thing about Docker is that this product enables the containers to communicate through the channel of transmission. Compared with virtual machine, Docker containers are more lightweight, because Docker operation strategy is centralized by a single operating system kernel. It uses images to create containers. Images which specify the exact content of containers individually. They are the result of merging and modification of standard images downloaded from public repositories. [33], [34]

There are, however, container orchestration platforms. Containers are ephemeral, they come and go based on the need, therefore there must be some software managing this coming and going especially in large, dynamic environments. Container orchestration is to properly manage the lifecycles of containers. This software supports the following tasks:

- Creating and deployment of containers within the dynamic environment.
- Provisioning the redundancy of containers.
- Rescaling containers or removing them for balancing purposes such as evenly spreading application load across host infrastructure.
- Moving the containers between hosts for provisioning purposes or when a host dies.
- Resources allocation in a dynamic way between containers.
- External exposure of services running in a container with the outside world.
- Dynamic Container load balancing of service discovery.
- Visibility and health monitoring of containers and hosts.

- Online configuration of an application for the containers running it. [35]

Container orchestration software adds containers and connects information about repositories and networks so that increase scalability and functionality of applications. Along with this, it authenticates the process of accessing containers and keeping components separated from one another, just by setting authentication requirements. Container orchestration software also enables developers to allocate simultaneously multiple containers for implementation within applications. Moreover, it lets the process of running instances automatically run, links containers and provides hosts. [36]

Container orchestration products necessarily have the following attributes:

- Enable administrators to provision hosts.
- Schedule dynamically and automate Deploying container.
- Allow running instances of multiple containers.
- Health monitors the container and alerts the user of failed container. [36], [37]

2.7.1 Kubernetes and Main Components

Kubernetes is designed, for providing container orchestration system, dealing with containerized applications. It was first made open-source By Google, where containers are arranged in groups, to transform an application into logical units, which are easily manageable and discoverable. Practically it helps automate deployment, scaling and management and also operation of application containers all over the clusters of all hosts. It supports many products of containers, including Docker. Therefore, many companies providing services, take it as the basis for constructing their services, on which almost all type of cloud native services can be introduced. Also, many software companies adapt themselves with this container-orchestration system. [37], [38]

Container technologies offer many benefits to service providers and in particular to developers, in fact, container technologies totally changed writing and deployment of applications. Developers can put all parts of an application, even its libraries in one package and ship it out, it enables them not to set any dependency on virtual machines.

Containers and the tools to utilize them, cannot handle the situations in which complex applications are interacting by their components. In these scenarios there must be an independent part managing the containers and their deployment, so that meet the required elements of the orchestration and this is what Kubernetes is about. [36], [37] Kubernetes orchestrates the containers, by which the applications are containerized. The most important benefit of Kubernetes is that, in the situations in where many complex

applications are deployed simultaneously, it can help handle the orchestration of containerized applications. Another important benefit of Kubernetes is allowing easy container management, therefore allowing horizontal autoscaling for the pods, suiting both stateless and stateful app, allowing customization and supporting pre-built solutions to run the app an open-source orchestrator and etc. [39]

Kubernetes has many different features and components however, for this study only the ones that are important for this thesis will be briefly discussed.

Namespaces are some sort of partitioning of the resources, which Kubernetes manages into non-overlapping parts. They are designed for better handling the situations with heavy request traffic of many users across the network. [36], [37]

Kubernetes presents its most basic unit for grouping containerized components as a **Pod**. By grouping containerized components, a pod comprises several containers which are going to be located in a pairwise way on the host machine for some reasons such as resource sharing. A unique IP address is allocated to each pod preventing from any further conflict in application running. Moreover, it enables a container to reference another one within the same pod or even different pod. In order to manage pods, Kubernetes API or any independent controller can be used. [36], [37]

Kubernetes needs to update and run its Pods, therefore a controller is needed to check for updates. Deployment controller goes for declarative updates for these two parts of Kubernetes, in order to add to the flexibility of system. For instance, the desired state in a **Deployment** object could be achieved by a transition from actual to desired state at a rate under control. In practice it can create or remove applications. [36], [37]

Kubernetes has an API object managing external access to the services in a **cluster** which is a simply a collection of machines or workers together in a group under one Kubernetes control plane. This means access from another cluster to a service in a certain cluster.

A set of pods interacting with each other and are defined by a label selector create a Kubernetes **Service**. For the services, service discovery is carried out by using environmental variables or using Kubernetes DNS. A service can be exposed inside a cluster or can also be exposed outside a cluster. [36], [37]

Pods in Kubernetes handle batch processes, batch processes are kind of processes taking a while in order to complete certain operations such as backup or restore. There must be an independent part to supervise the Pods, this is what is designated to **Jobs**. More specifically, this is like dividing, Jobs create certain number of Pods to complete the task. When the task is completed, one by one the Pods get completed. Finally, after

reaching a certain number of successful Pod completion, the Job is complete and just by deleting the job, all the corresponding Pods will be cleaned up. A Job can also run several Pods in parallel. [36], [37]

Kubernetes **Ingress** typically is HTTP. Ingress help offering name-based virtual hosting, load balancing, and SSL termination. In a better sense, Ingress enables a mechanism which allows clients to access a service outside the cluster. [36], [37]

In order for locally running Kubernetes, we can typically run a single-node Kubernetes cluster inside a virtual machine. **Minikube** is the tool for the task and is suitable for testing and development purposes. Also, Minikube can create cluster for us and also supports features including: port forwarding, **ConfigMap** which is set of configuration for the application and used by Kubernetes, **Secrets** that is a secure way to handle password and sensitive information used by Kubernetes as well, Dashboards, DNS, container run time, Ingress and finally container network interface. There are some limitations to using Minikube than real Kubernetes deployment, for example, you will always get a single node (master) cluster and many of the internal networking is totally done in background and cannot be modified or seen properly. [36], [37]

Since Istio is the chosen service mesh framework for the work carried out, few of the essentials and configurations specific to this platform is discussed as well to better understand the logic behind it.

Kubernetes pods are ephemeral, meaning when they die, they are not recoverable. However, considering a Kubernetes service as a system providing a logical set of pods and corresponding policy to access, one can see nodes in a Kubernetes cluster run their individual KUBE-PROXY each stands for implementing a virtual IP for services. **VirtualServices** and **DestinationRules** are similar in a way used in different places which are a set of user-defined rules for routing the flow of basic unit of network, here containers or a group of them to the destination in a dynamic manner. [4], [37]

Helm is an application package manager that tries to make Kubernetes and its objects such as ConfigMaps, services, pods, persistent volumes less complex by proper management. As a package manager, Helm packages all the thing in the form of one application. It helps performing the deployment easily, less complex, standardized and reusable. [40]

2.7.2 Operation and Monitoring Tools

To evaluate the reliability of services, performance or debug the potential problems one can address Kubernetes cluster by monitoring. In other words, detailed information about

the application resource usage will be available for assessing the performance of the application. Therefore, application monitoring depends on multiple monitoring solutions. It means there are several ways to collect monitoring statistics of clusters each way is dependent on its own separate pipelines.

One way to monitor the applications is to analyze the logs of the running apps and containers. Elasticsearch, Logstash, and Kibana (**ELK stack**) or Elastic stack as monitoring tool, are responsible for the trade for logs aggregation and analysis. [41]

A typical scheme of ELK where different sources of events are collected, originated, searched and visualized is as follows in Figure.6.

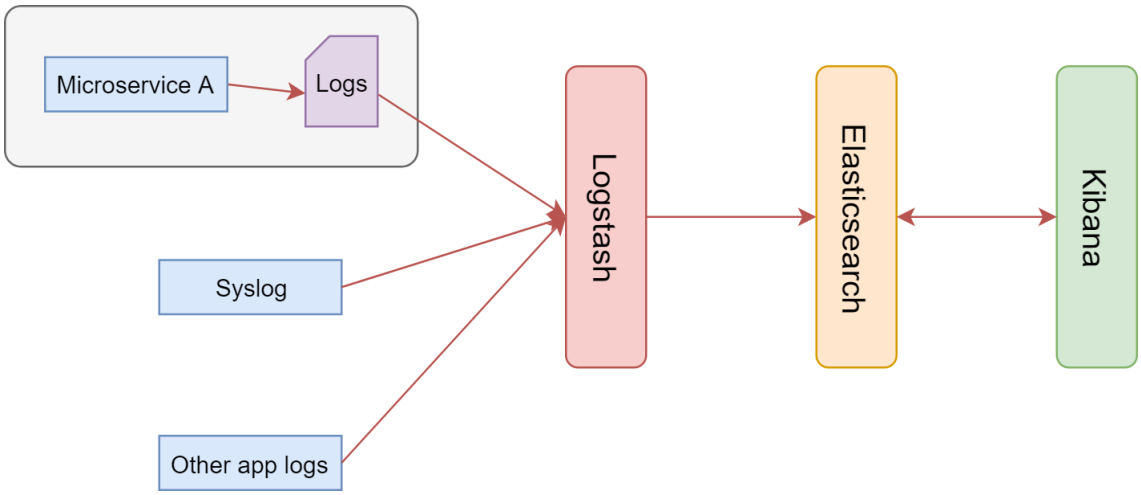

***Figure 6.**: Elasticsearch, Logstash, and Kibana, approximate scheme.*

Another means to monitor and analyze the applications is the need for Tracing. Request tracing is the final strategy as a monitoring tool which enables tracking operations inside different systems along a timeline. It is specially designed for tracing requests spans in distributed systems.

Zipkin and **Jaeger** are two famous request tracing object tools, build for huge organizations. Zipkin was built by Dapper and later developed by Twitter and Jaeger was developed by Uber. While both of them have similar architecture designed to send traces to the trace controller, they have some basic differences. The trace controller gathers the data and analyzes the relevancy between the traces. Later a UI is provided to further inspect the traces. An important decision-making parameter for making a difference between the objects is the languages and libraries each supports. In this sense of supporting many languages, Jaeger is preferred due to supporting most of the common languages. However, Zipkin seems to support more libraries. It is notable to mention that Jaeger supports any open tracing instrumentation library, therefore it is expectable to

take over this realm in near future, but Jaeger potentially stronger since it works with any open tracing instrumentation library. But a wise decision is to use both. Since Jaeger supports Zipkin's API, we can choose the instrumentation libraries from Zipkin and collector from Jaeger. [42], [43]

Prometheus as a metrics platform was introduced to handle heavy loads of events and logs. It stores and analyses various type of data from IoT sensor data to DevOps monitoring which can be used to be sent and used by a visualization tool to look for the desired data. In order to enable the system to interact with a unique query language in embedded inside Prometheus as an open source monitoring object. [44]

The Kubernetes provided **Grafana** as a monitoring tool for Kubernetes cluster's performance. With its dashboards, Cluster, Node, Pod/Container, and Deployment it supports to visualize the required Prometheus exporters automatically. Grafana gathers information including status of high-level cluster and node and also status of lower level pod and container respectively for alarming and troubleshooting. [45]

Another helpful and well utilized tool during this work is **Kiali**. Kiali is deployed to Kubernetes as a monitoring tool which cooperates with Istio. Therefore, Kiali deployment enables activating many features of Istio for monitoring purposes. Kiali is another monitoring tool but designed specifically for Istio. The main benefit for using this tool is its graph type visualization for the deployed functions, their connectivity, traffic distribution between functions. [46]

3. RESEARCH METHODOLOGY

In this chapter, along with the background presented in the previous chapters, we are aimed at thinking about the solution and discussing the solution so that cover all important aspects of it. Since the goal of this work is to consider SBA with the service mesh platform in the context of 5G core, this chapter deals with providing the architecture based on services in order to best fit the expectation of 5G network and its requirements. Until now, different products have been introduced and talked about, and now we stand for making the decision on how to use one or more than one of them to form the optimum solution. There must be a pairwise comparison between every Kubernetes service mesh to figure out the most preferable one.

3.1 Service Mesh Technologies

Based on the information provided by second chapter, and with an emphasis on Kubernetes, one of four most well-known service meshes including Linkerd, Linkerd2, Consul and Istio is going to be selected. Thanks to publicly available comparison made by Kubedex and people's contribution firstly, we can have a look at the features individually in the Figure. 7.

	Istio	Linkerd	Linkerd2	Consul Connect
Model	Sidecar	Node Agent	Sidecar	Sidecar
Platform	Kubernetes	Any	Kubernetes	Any
language	Go	JVM	Go / Rust	Go
Protocol	HTTP1.1 / HTTP2 / gRPC / TCP	HTTP1.1 / HTTP2 / gRPC	HTTP1.1 / HTTP2 / gRPC / TCP	TCP
Default Data Plane	Envoy (supports others)	Native	Native	Native (or Envoy)
Sidecar Injection	Yes	No	Yes	Yes
Encryption	Yes	Yes	Experimental	Yes
Traffic Control	label/content based routing, traffic shifting	Dynamic request routing, traffic shifting, per request routing	No	static upstream, prepared query, http api / dns with native integration
Resilience	timeouts, retries, connection pools, outlier detection	timeouts, retries, deadlines, circuit breaking	Retries, timouts	Pluggable
Prometheus Integration	Yes	Yes	Yes	Yes
Tracing Integration	Jaeger	Zipkin	None	Pluggable
Host to Host auth	Service Accounts	TLS Mutual Auth	Experimental	Consul ACL
Agent Caching	Yes	No	No	Yes
Secure connection outside cluster	No	Yes	No	Yes
Complexity	High	High	Low	Low

Figure 7. *features of four service mesh options for our solution.* [47]

Linkerd is the first product introduced as a service mesh and was expected to excel other products. It is used on DC/OS on Kubernetes. Anyway, later it made the Kubernetes suffer from some problems relating to web socket, TCP request, and overhead. This

product does not handle TCP requests and web sockets Since, it shows good performance in handling one node agent per host. However, Kubernetes goes quickly toward sidecars of per pod proxy, therefore Kubernetes have to deal with too much overhead each time.

Linkerd2 is very similar to Linkerd but the language is different. It is written in Golang and Rust which help to solve some problems such as those of memory. While there may be problems similar to Linkerd, there are some advantages such as supporting many protocols or low data plane latency. After all, it has simple configuration. One important thing which is implemented in Istio but for Linkerd2 still at the planning phase is tracing in a distributed perspective. Another thing is stability, in fact this is not the most stable product compared to some products such as Istio. Along with all advantages and disadvantages, one can say the most interesting feature of Linkerd2 is its simplicity, and if the product in use is Linkerd2 while appeasing all requirements, it can be a good choice for future services. [4], [48]

The last version of Consul includes data plane and control plane written in Go. In situations that not all the objects are migrated onto Kubernetes and there are still services in VMs, this may be a good choice. But generally, connect have some disadvantages such as having no centralized control plane. Moreover, due to the separation between layer 4 and 7, the data layer would bifurcate. But Istio uses Envoy as data plane so that supports more attributes form layer 7. Consul has simpler configuration compared with Istio, however it does not support many features. [47]–[49]

Istio compared to other products is the one which is pioneered many new and important features such as stability or special sidecar injection. Istio is not the very first service mesh produced for the market however it is the very first in offering new features for better supporting different systems with their corresponding configuration and request traffic level. One of the highlighted negatives about this product is its complexity, and in particular complexity in configuration. Istio tried to address its complexity by providing appropriate defaults. In many cases even by setting the default configuration, one can find Istio as the most flexible product in the market. It takes relatively short time to run on Minikube and Istio in a PC or laptop as a test stage. Another bright point about Istio is many online materials about step by step launching a system working with Istio. Having Kubernetes as the infrastructure of the system, it can be concluded that generally the best choice is Istio noting that the choice is in relation to this thesis purpose and can be something else for a different use case.

3.2 Proposed Solution

Here Istio together with Envoy is what would be preferred as the state of the art solution. As previously stated, while it is not the first open source network proxy used as a service mesh, it is pioneer in many new innovations in the field. It is complex but very powerful and supportive. Istio is also considered as the best service mesh by many experts or companies such as Google. Istio paired with Envoy offers a universal data plane supporting diversified services in 5G network. Along with Istio, Envoy is the best choice as the universal data plane.

Istio, as already mentioned, usually pairs with Envoy by default, and provides a control plane to handle various service proxies. Some experts consider Istio as the best or at least one of best service meshes that have been introduced

With the 5G network, it is important to guarantee some features such as network visibility, services mutual communication, load balancing, and extensibility. Istio has predicted these expectations and provide the best possible response to them. Briefly it supports balancing of different traffics load, controlling and routing the traffic, strong security in cluster service to service data transfer and etc. Istio allows for uniform privileges across a network of request based services. Therefore, with the uniform availability of many features, Istio enables the network to get closer to multidimensional optimization. It means that many aspects could be taken into account.

3.3 Istio

Following is a brief overview of the core features provided by Istio:

Traffic flow control

Istio is designed and developed considering the high flow of traffic and the urgent need to routing it. It goes for setting configuration dynamically routing the traffic and API simple procedure of configuration and splitting traffic lead manage and response traffic and API calls. Istio makes simple configuration in a level of service so that split the traffic into equal percentile to respond in parallel and at the same time monitor the traffic and corresponding fluctuations.

Security

Service mesh must provide the facilities for high security in any level of communication, in this regard, different service meshes prepare a different level of security for developers. Since Istio makes it available a very high level of security in most of the levels by securing channels, and initial authentication and logging in either by services or persons,

it allows the service owners to have enough time to concentrate on security in application level. Technically Istio is a service mesh not only provides services-mutual-communication but also at the same level worked on security of the communication, therefore the only task remaining for the developers is to think of application layer security regulations and runtime, such as those of between services or pods, while the is no need to rewrite the application. Finally, it is not independent of the infrastructure, but Kubernetes and Istio is the best pair of this time.

Visibility

Istio allows monitoring and tracing continuously which leads to a clear observation of service mesh moment by moment operations. Moreover, performance of the system and its impact on different objects, monitoring, and logging give you real time information about service mesh deployment.

Istio has a component namely Mixer, which provides real time policy control and telemetry collection in order to allow the best monitoring and control on service mesh interaction with the infrastructure. Therefore, Istio make it available to practically monitor and troubleshoot the system and subsystem parts.

Independency of platform

Istio is independent of platform and while the best pair could be made by Kubernetes, but it can work with various environments, including Kubernetes, Mesos, and other products. As mentioned Istio can be deployed on Kubernetes and in this line currently supports: registering and deployment of services on Kubernetes and also handles virtual machine individually. [4]

3.3.1 Traffic Management Between Proxies

A service proxy is considered the infrastructure reliable for the services of applications in order to exchange data with any exact destination. Therefore, it enables the network point to point communication using exact address allocation to any given destination. Istio uses a service proxy based on Envoy Proxy. It is fast and highly efficient Layer 7 (one of the layers of a communication network dealing with application) proxy coded in C++, allowing to handle millions of requests in a second. It balances the loads, monitors the health and gathers metrics of the application layer. Envoy is deployed as a service proxy very similar to the written code of application.

Data plane and a **control plane** are logical parts of a service mesh. The **data plane** is provided as sidecars that are originally the same as proxies. Istio uses Envoy as the set of proxies carrying out the data plane tasks. Istio with Envoy embedded inside is one of

best designs for networks relying on service mesh concepts and performances. Therefore, any communication between services is handled using proxies.

The **control plane** is the other part of service mesh figuring out the management of communications as the configurations, so that can guide the traffic by the help of data plane (proxies). The control plane also takes part in monitoring, for instance, support Mixer configuration and telemetry collection. In the Figure below both, the control and data plane can be identified.

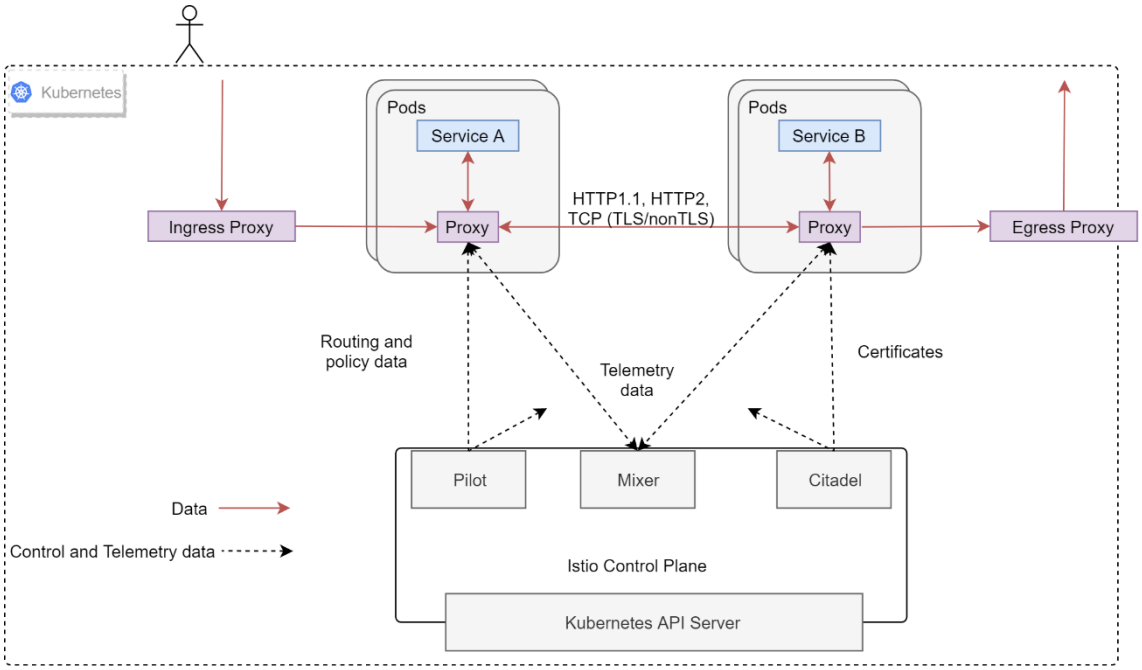

Figure 8. *Istio Architecture with Data plane and control plane virtual interaction. (From Istio* [4]*)*

3.3.2 Core Components

Istio like other service meshes has core components arranged to best deliver the services and handle the traffic request with the appropriate response. We can see in the following figure briefly how logical part of Istio cooperate to provide and deliver the best response.

Istio with **Envoy** as the high performance traffic proxies, support features including:

- Automatic service discovery
- Circuit breaking
- TLS termination Balancing loads and checking the healthiness of network
- HTTP/2 and gRPC proxies
- Terminating TLS
- Dynamic traffic splitting and routing

- Fault injection
- Rich metrics

Istio deploys Envoy as **sidecars** to the service and needs no new pod for Kubernetes. Under a good embedding of Envoy, Istio exploits extracted signals about traffic behavior and feed to Mixer exerting regulations, and later to visibility systems to infer knowledge about the performance of the mesh in whole.

Service discovery or **Pilot** is an integral part of Istio dealing with Envoy sidecars, so these discovery services and manage them. It also carries out shaping and guiding the traffic request along with control over capability and resiliency of the mesh network.

Pilot sets and alters traffic routing regulations into a new configuration of Envoy type. These routing rules manage the traffic behavior and considering the runtime, well propagates the traffic which is performant in load balancing. Pilot standardizes the service discovery mechanism so that any product conforming to Envoy structure, can use the mechanism for the dynamic service discovery. One benefit about Istio running infrastructure is that it uses the same interface for network management control operator to control the traffic regardless of simultaneously working with a couple of Kubernetes and Nomad and Consul. This is because Pilot help to support multiple infrastructure utilization while using the same interface. [4]

One noteworthy aspect to consider here is another issue is the use of the 5G Core components like NRF. As its primary tasks, it maintains a list of NF profiles of available NF instances, their supported services and service discovery function which is similar to what the Pilot in Istio does but in a higher level and the coexistence of the two functions must be addressed and seen with the future 3GPP releases at the moment. Nevertheless, since both the Pilot and NRF are not contacted as frequent as the NFs or any other application, they can both be there and work within the service mesh.

However, this can be addressed and in fact, is not an issue since they provide the discovery functions at different layers and can coexist.

The Pilot does manage the running and indirectly runtime of microservices regardless of what or how many environments underlie the network architecture. It provides an updated routing table at each moment to guide the containers and pods and services maintaining the current topology and be equipped with the updated RouteRules. The RouteRule is responsible for finely graining the request distribution. [4] Pilot has a specific architecture as seen in Figure 9, allowing for automating and flexibility of the network.

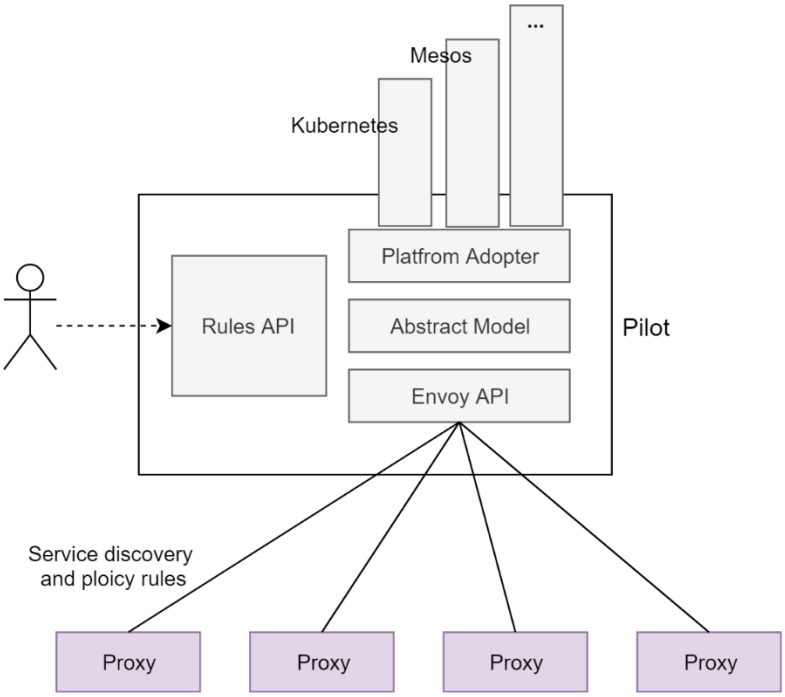

Figure 9. *Pilot architecture. (From Istio* [4]*)*

Several components take part in the shaping and management of traffic management. The most important of the is Pilot that is responsible for management and configuration of instances of proxy (Envoy) in the Istio. Using Pilot allows traffic routing rule selection for the flow between proxies and provides failure recovery. Practically it enables features like timeouts, retries and breaking circuits. You determine how to split traffic between Envoy proxies and failure recover failure. A standard way is maintained of all the services in order to make it possible for any instance to know about the others through the discovery service.

Intelligent distribution of traffic between destination instances is done by an unabated collection of load balancing information by the Pilot. Finally, lifecycle of instances is checked and controlled by Pilot. Its canonical representation of services is independent of the platform. It is available by Platform-specific adapters in Pilot which populate this standard modeling appropriately. Based on this, a specific configuration will be made by Envoy which is responsible for further communication and maintenance of network status. Pilot provides all features of service discovery, dynamic updates to load balancing pools and routing tables and enable any available kind of traffic management by its rule configurations.

Mixer is another important component of Istio dealing with access control and using policies related to the microservices management across the service mesh. This is a component operating in a platform-independent way checking polices and receiving telemetry data. Its interaction with Envoy proxies helps it to maintain an up to date status of

telemetry data and leads to making better evaluation by sending service level features to Mixer. This model allows Istio to interact simultaneously with various infrastructure such as Kubernetes in the backends.

Mixer is the Istio component for mixing data objects or in other words bringing things together. All telemetries of each of the applications will be sent to Mixer for evaluation and it enables Mixer maintains the standard logical structure of the usage and access policies for the overall suite of microservices or pods. Some controlling actions such as rate-limitation or creating and monitoring consumer metrics can be done by Mixer. Its pluggable backend architecture is essential for further extensions and advents of new capabilities for more assignment. [4]

Istio is designed in a way that collects telemetry data to better evaluate the network different parts status. Through tracing, monitoring, and logging which is done by its components, continuous monitoring of the service mesh performance is maintained. Performance of any service is monitored and a good understanding and insight of objects performance and impact on other thing are obtained. Visibility over network and real time monitoring helps provide good telemetry data which is used by Telemetry component to control the performance and the direct and indirect effects of each part on others. As already suggested, policy control and telemetry are the tasks which Mixer is responsible for. Therefore, other components will be insulated from deployment or implementation details

These are the attributes allowing effective setting, monitoring, and enforcement of service level objectives (SLOs) on services. Eventually, detection and fixing the issues is easily performed.

Citadel is the component supporting authentication either for services mutual communication or data transfer with the user at one end and responsible for key management. In order to make up to date the unencrypted traffic within the mesh, Citadel has certain regulations. Once the service identity is subjected to the polices, Citadel can be used to insert the polices. Using Citadel authorization capability, operators can control access to the services for any individual. Furthermore, Citadel enforces health checking process, therefore it is a great help for monitoring and fault finding.

Citadel has a health checking capability supporting some features such as signing services and is optional in the deployment of Istio (That can be optionally enabled). The health checking feature is able to detect the failures of Citadel CSR signing service, by periodically sending certificate signing request to the API. It is continuously developing for better health checking and corresponding actions.

Citadel uses a prober client module for periodical checking Citadel's status. While we have a healthy Citadel, the modification time of the health status file will be updated by the probe client (always empty file). Otherwise, nothing will be done. Citadel relies on a K8s liveness and readiness probe with command line to check the modification time of the health status file on the pod. If the file is not updated for a period, the probe will be triggered and Kubelet will restart the Citadel container. The Auth is another Istio's component, which is worthwhile to be noted here. In practice, it stands for some services for certificate such as signing, issuance and revocation/rotation. [4]

Galley goes for validation of user authored Istio API configuration for all other components. It is expected that Galley becomes responsible for configuration procedure, data processing and allocation component of Istio. It will stand for separating the other Istio components from the issues related to the details of attaining user configuration infrastructures such as Kubernetes. [4]

Istio has the capability of **load balancing** as a service mesh. This service mesh provides other capabilities like management of traffic and its control, service discovery, providing resiliency, visibility, security, and etc. Istio has the components responsible for visibility and monitoring, health check, issue detection, and continuous telemetry data collection. this is why the Istio enable load balancing in the network because those features support dynamic traffic routing or splitting.

In service mesh, services can access each other individually by DNS names, and distribution of traffic is executed across the instances and pools for load balancing via complex algorithm for the tasks. These algorithms for load balancing are performed under three modes; round robin, random, and weighted least request.

Along with these algorithms for load balancing, Istio proxies (Envoy) perform a periodic health check for instances and send the data to Mixer. For checking the health, Envoy uses a circuit breaker pattern to evaluate healthiness, unhealthiness or according to health check of API call. A pre-specified threshold is exerted to the number of failures of health check in order to decide whether eject or not eject from the load balancing pool. [4]

Istio is deployed in a way that easily performs rules configuration and routing traffic leading to performant routing and splitting the flow of request traffic of services and API calls. While Istio is not the one having simplest configuration among all products of the service mesh, it provides circuit breakers time outs and retries that simplify the service or application-level configurations.

Continuous observability into network traffic flow, and unusual failure recovery features, prevent causing problems by issues happening in the network. These attributes detect the potential cause of problems, making connections almost fault-free, and your network more performant regardless of the conditions network faces. [4] As seen in the Figure. 10, Istio can manage the traffic based on either the content of the request and reroute it to a specific instance of an application or even based amount of traffic sent and handled by the network to those instances.

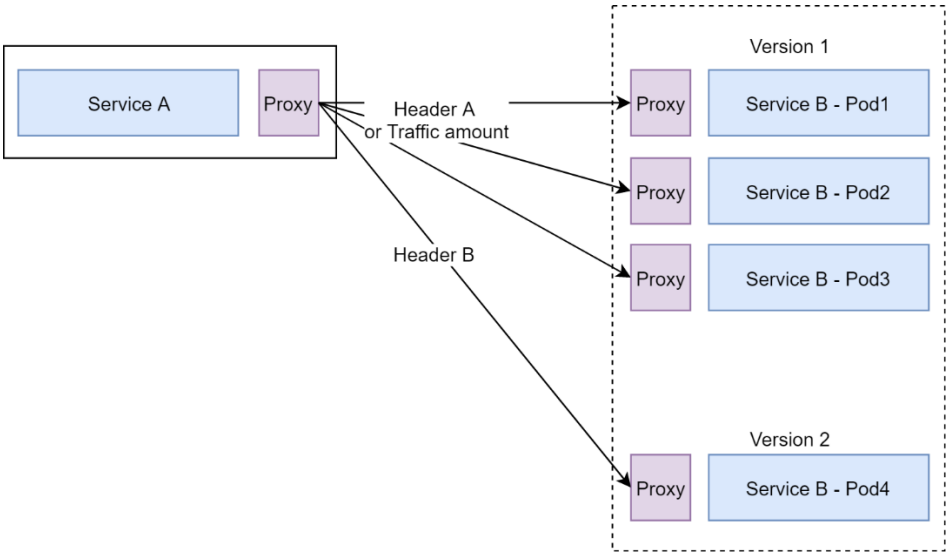

Figure 10. *Istio traffic routing and management, content or traffic wise. (From Istio* [4]*)*

Traffic Management of Istio has specific features. Independency of traffic from infrastructure scaling enables Istio to prepare various traffic shaping attributes that are independent because operate out of the application. Request routing is another performant feature of Istio which is done by the standard existence of services maintained by Pilot. In fact, continuous monitoring along with Pilot-specific tasks is performed to make a deep insight into the service mesh and simplify the traffic management and request routing. A new concept called *service version* is provided by Istio split service instances by versions (v1, v2) or environment (staging, prod).

Istio is designed and implemented as a service mesh handling service to service communication and request traffic of many types. Considering the fact that traffic toward and traffic outward the service mesh is handled and routed by Envoy proxies, Istio uses a deployed Envoy proxy in front of services in order to conduct A/B testing and deploying canary services leading to better service delivery to users. Envoy sidecar enables failure recovery attributes. It includes items such as timeouts, retries, and circuit breakers and

obtain detailed metrics on the connections to these services, the only thing needed is to redirect the traffic to external services which is optional. [4]

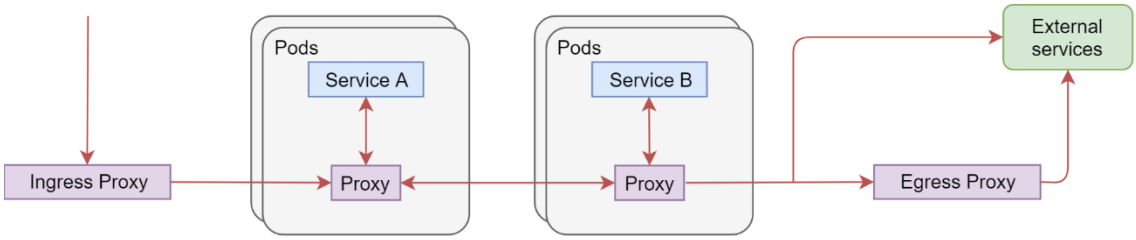

Figure 11. *Request flow for Ingress and Egress trafficof a service. (From Istio [4])*

- Control Egress Traffic

Usually, Istio's services only can reach out to URLs inside of the cluster. This is because the pod redirects outbound traffic to the proxies of sidecars leading to only supporting addresses of intra-cluster. This task stands for guiding Istio to expose external services to Istio-enabled clients. [4]

- Control Ingress Traffic

With the Kubernetes as infrastructure, the Kubernetes Ingress Resource detects the services being exposed to outside the cluster. Istio as a service mesh on any given infrastructure utilize Istio Gateway for the task, which is a different configuration model dealing with cluster entering the traffic flow. A Gateway allows Istio features such as monitoring and route rules to be applied to traffic entering the cluster. [4]

Istio manages traffic by bifurcating traffic flow and scaling environment, Pilot helps collect data to better look into the traffic flow and finally traffic will be responded to by the rules to be followed by traffic, rather than the which specific pods/VMs should receive traffic. For instance, you can configure via Pilot that how many percents of traffic for a particular service to go to a canary version irrespective regarding the size or send traffic to a version which is content-dependent.

Generally, clients know nothing about the different versions of the service. Here is the place that there must be a guaranteed way to access. Proxies of Envoy take part to handle the system, they are authorized to reach out to the services by their hostname/IP address. The Istio sidecar/proxy (Envoy) faces the requests and forwards all data in a client-service communication.

Envoy uses Pilot and its routing rules to realize the actual version of service, which leads to the fact that application code becomes independent of changes and development of services and clients gain what the need regardless of the evolution of services. Istio

selects the version via its Envoy considering headers and source/destination tags or the weight of each version.

Istio is capable of multiplying the instances of certain service version. It is done by balancing the load and service discovery. While Istio does not prepare a DNS, applications use DNS service of the platform such as Kubernetes DNS for resolving the service IP, based on names.

3.4 Technical Implementation of Project

Here we concern with the technical implementation of the project which includes how to technically implement Istio as our selected service mesh with Envoy to support the proxies. Istio has the capability to be expanded and updated by new features either in its control plane or data plan. It makes Istio and Envoy the very good couple to serve as the network service mesh even in networks supported by Google or other web-scale companies. As a trial, we implement Istio on a PC as a server supporting many computers in its network. All coding and configuration will be done by the help of information provided by Istio website.

4. EXPERIMENTS AND RESULTS

4.1 Testing in Different Environments

There are two main fundamental environments to deploy Istio, one with Nomad and Consul which in fact runs all the components with Docker containers, and the other method is with Kubernetes cluster which is the primary one.

One key element worth mentioning with Istio is that as for the envoys to carry upstream traffic for services, the traffic must be HTTP/1.1 or HTTP/2, otherwise, they don't connect to the HTTP/1 services.

First tested environment was to deploy Istio service mesh using Docker compose which runs every component and service like Pilot, API server, etcd (database), Zipkin and etc., therefore the application that is going to use the service mesh also needs to be deployed using Docker so even the application's parts (microservices) will be run in containers like the sidecars for each one of microservices' application and itself.

To use deploy the service mesh with Docker, we still need to have the Kubernetes command-line interface for managing the Istio API resources.

Since the consul-based environment of Istio is meant to be used for testing purposes and scenarios to view the functionality of the platform, many of the features, for example, the Policy checks, Ingress, Mixer, Citadel, Prometheus are not present in this setup, that's why in case of an error in one part it is a very difficult task to debug the problem and the containers.

One issue with using Docker is that there is no understanding of Pods in this environment, consequently, the sidecar needs to run in the same container with the application. In order to do this, it uses a service called Registrator made by Gliderlabs which takes care of the automatic service instances' registration to the consul-server which is also deployed as a container in Docker.

One major problem with this environment was that it was extremely slow compared to a full setup on Kubernetes, where 10s of seconds of latency could be observed to get a response from the application and therefore some of the microservices would reach their timeout limit and wouldn't retrieve the requested output.

The next tested environment was installing Istio components on a Kubernetes cluster which in truth is the primary and intended method for deploying Istio service mesh with all the parts.

Since Kubernetes is a part of the Cloud Native Computing Foundation – CNCF, it is widely adopted by many major cloud providers like Amazon Web Services, Microsoft Azure, IBM Cloud, Google Kubernetes Engine, Alibaba Cloud and OpenShift. However, to run Istio in a local environment virtually, we use Minikube.

Starting Minikube has some parameters and one of importance is the VM-driver. This parameter indicates where the Kubernetes cluster components will be installed. for example, with `--vm-driver=none` the components will be on the host using Docker containers, but with `--vm-driver=virtualbox`, instead of host, they will be in a Virtual Machine.

There are other drivers to be used but for our testing purposes and being able to see the logs from different containers we used the none driver mode because with other drivers like VirtualBox the background components are not visible and easily accessible.

After setting up the cluster, to install Istio there are few different options to setup the Istio's control plane and each one is suitable according to the needs and application scenario, for example, such as using the provided scripts either with or without the TLS authentication between the envoy proxies. There is also the option to deploy Istio using Helm templates to create a Kubernetes script where it is possible to set many different parameters regarding the core components to get a desired tailored framework according to the application in mind.

This approach is the recommended way to install Istio for using it in a production environment because of the very high level of config customization to the control plane components and proxies to the data plane.

To point out some of the configurations which can be modified we can change whether the ingress should be enabled or not with `--set ingress.enabled=false/true`, or whether the automatic sidecar injection must be enabled or not with `--set sidecarInjectorWebhook.enabled=false/true` and alike with Pilot, Mixer and other components, and the platform will still work fine because the Istio has been designed in a module independent way that can work and provide the traffic management functionalities for the applications.

All the different methods mentioned above to deploy Istio have been tested. The Consul method is not a very usable option as for the problems explained and not being able to debug problems easily. In Kubernetes, in first attempt, Istio was installed using the provided script without the TLS authentication which was a success. The Helm method was

also tested with the next attempt to use the different configurations while deploying Istio, like with no Mixer Policy, Ingress or Automatic Sidecar Injector in different orders to study the robustness and how it affects the latency with applications.

The next method to try out was the TLS authenticated provided script, where it was installed on the host machine rather than VMs for better performance. The TLS authenticated Istio was the chosen methods because of the nature of our application, 5G Core components as they are defined to be TLS protected according to 3GPP standards.

It must be noted that in the configuration file, the proxy concurrency value was set to 0 which means utilizing all the CPU assigning a thread per proxy also helping with reducing memory use.

After successfully deploying Istio on Kubernetes, all the core services were running as seen below showing their service address and ports they are running on.

```
   test@VM:~/Istio-1.0.1$ sudo kubectl -n Istio-system get svc
   NAME                        TYPE           CLUSTER-IP        EXTERNAL-IP    PORT(S)
AGE
   grafana                     ClusterIP      10.108.226.35     <none>         3000/TCP
4m
   Istio-citadel           ClusterIP     10.96.46.73     <none>       8060/TCP,9093/TCP
4m
   Istio-egressgateway       ClusterIP     10.103.212.92    <none>       80/TCP,443/TCP
4m
   Istio-galley            ClusterIP     10.102.117.183   <none>       443/TCP,9093/TCP
4m
   Istio-ingressgateway               LoadBalancer      10.101.107.232      <pending>
80:31380/TCP,443:31390/TCP,31400:31400/TCP,15011:31561/TCP,8060:32551/TCP,853:30981/T
CP,15030:31149/TCP,15031:30583/TCP   4m
```

Program 1. *Istio specific services running on Kubernetes*

All the application pods which are deployed in the Istio service mesh, need to also have an appropriate proxy inside which must be inserted either manually with Istio command-line interface or use the Istio sidecar injector function which inserts the envoys as a separate container within that pod.

The main difference between manual and automatic injection is that automatic, the injection happens at the pod level creation so for example, deleting an existing pod which doesn't have the proxy container in it would be recreated with envoy containers attached. With manual injection it requires making adjustments to the deployment and restarting since all the pods must be restarted.

Essentially how the sidecars work is that every microservice has its own sidecar transparently sitting in the pod of the application, and it does not even have to be aware of its existence. The proxies capture the traffic and forward it to the next hop accordingly.

Kube control (kubectl) command line can be used to check all the pods running on the default namespace which is where the applications are deployed. As can be observed from the output below, all the pods are in Running status with 2 containers each which one belongs to the injected sidecar and the other to the microservice's container, whereas if the automatic sidecar injection is not enabled, we would get 1/1 containers ready.

```
test@VM009793:~/Istio-1.0.1$ sudo kubectl get pods
NAME                                      READY     STATUS    RESTARTS   AGE
details-v1-6865b9b99d-hpdmc               2/2       Running   0          5d
productpage-v1-f8c8fb8-z6zmz              2/2       Running   0          5d
ratings-v1-77f657f55d-wsbr5               2/2       Running   0          5d
reviews-v1-6b7f6db5c5-ppc8r               2/2       Running   0          5d
reviews-v2-7ff5966b99-7jk96               2/2       Running   0          5d
reviews-v3-5df889bcff-hgrn6               2/2       Running   0          5d
simple-webserver-75d95b858b-v898s         2/2       Running   0          5d
simple-webserver-redis-c55687cf5-9wqqs    2/2       Running   0          5d
```

Program 2. *Bookinfo application pods on Kubernetes*

The pod can be examined also by the Kubernetes user interface through the Minikube address, where as shown in the Figure. 12, the Productpage's pod has two containers, Productpage and Istio-proxy.

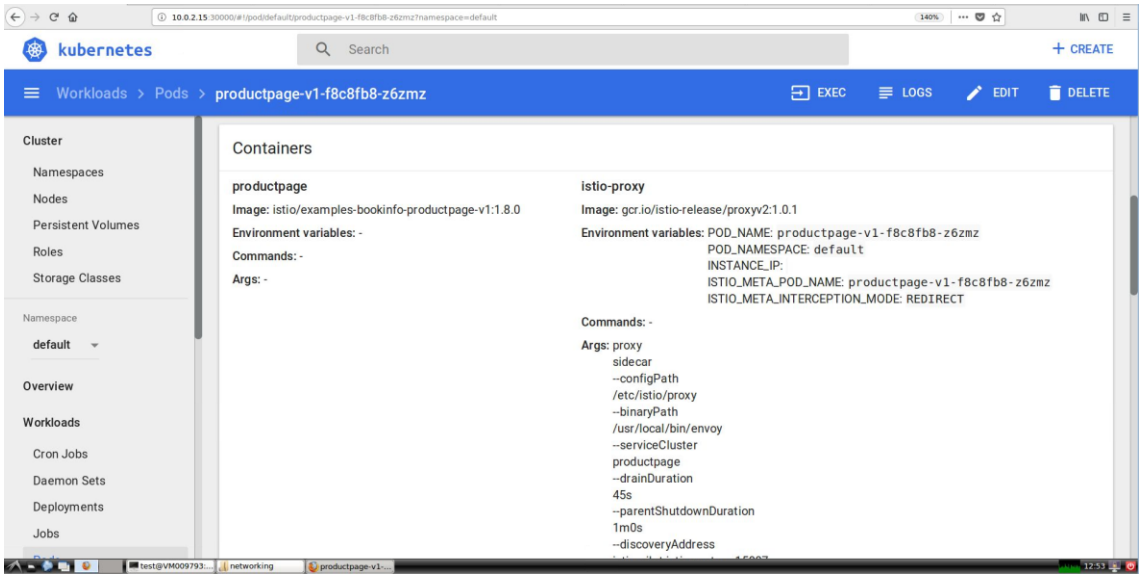

Figure 12. *Kubernetes Dashboard UI.*

4.2 Actual Setup and Test Application

When setting up the Minikube, the recommended number of CPUs are 4 and 8192 Mb of memory and for our testing purposes and due to different application needed to be tested we increased the resources assigned to Minikube to 10 CPUs and 14 Gb for the memory.

Before deploying the actual 5G Core components into Kubernetes and Istio, a few different applications were tested for functionality-wise and other aspect testing and encountering the problems that could possibly be faced.

1. Simple Webserver:

A very simple webserver application which is written with Go language with permissive free software licensed under the MIT license. It consists of a frontend and a backend Redis database for fetching some values.

There are various methods to deploy the webserver like Docker containers, Kubernetes or even as a native application. Due to the environment setup, Kubernetes option works very well with separate Deployments and Services for the webserver frontend and backend database.

2. Bookinfo:

Conductive to understating and to get a better view of how the services work, there is a well-known sample book info application in the Istio territory. It consists of four microservices which will help to a better interpretation of the features.

The four microservices of this application are:

- Productpage: This microservice written in Python can be considered as the root of the application since it makes the calls to the other microservices. – details and reviews.
- Reviews: Written in Java and includes the review information and can make calls to ratings.
- Ratings: Written with Node.js and has star ranking information on the reviews.
- Details: Written in Ruby which has the book details like the Publisher, Language and etc.

The reviews are written in a way to be able to demonstrate how Istio handles microservices with several instances, therefore there are three versions of reviews:

- v1 is only a simple review as text.
- v2 is the review and in addition, calls the Rating microservice and displays rating stars in black.

- v3 is the review and in addition, calls the Rating microservice and displays rating stars in red.

Some of the main reasons for the Bookinfo to be a good example to showcase Istio is that this application is written in multiple different programming languages and does not limit Istio to specific languages. It can be normally also deployed and is not reliant on Istio. The total architecture of the application is seen in Figure. 13.

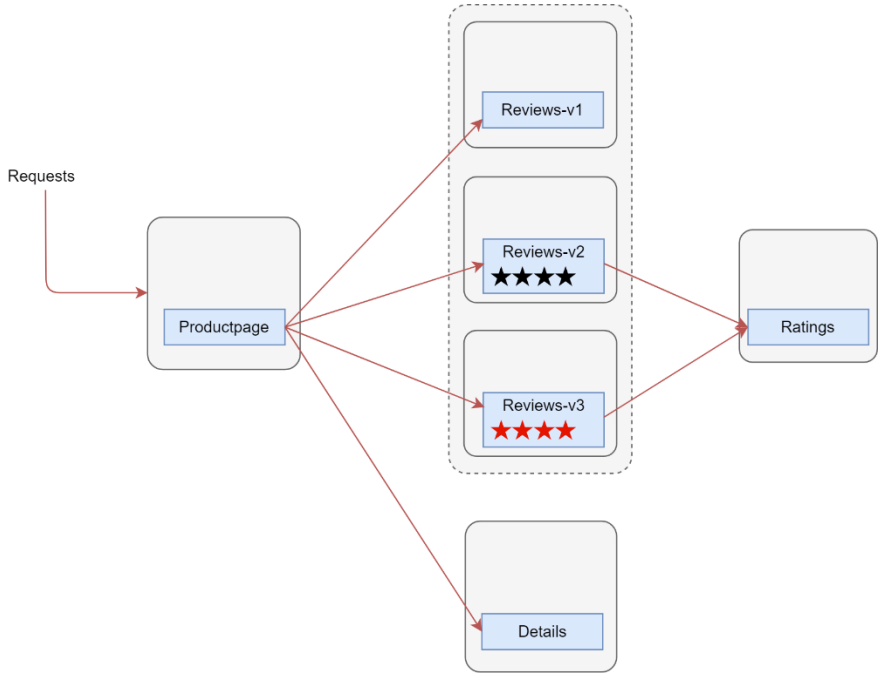

Figure 13. *Bookinfo architecture without its sidecars. (From Istio* [4]*)*

To get an Istio enabled application, it doesn't necessarily need changes to the application, what needs to be done is the application must be deployed in an environment where Istio is enabled and manage to inject the sidecar proxies into each of the application services which in the Bookinfo application, the overall deployment structure would look as illustrated in Figure. 14 below.

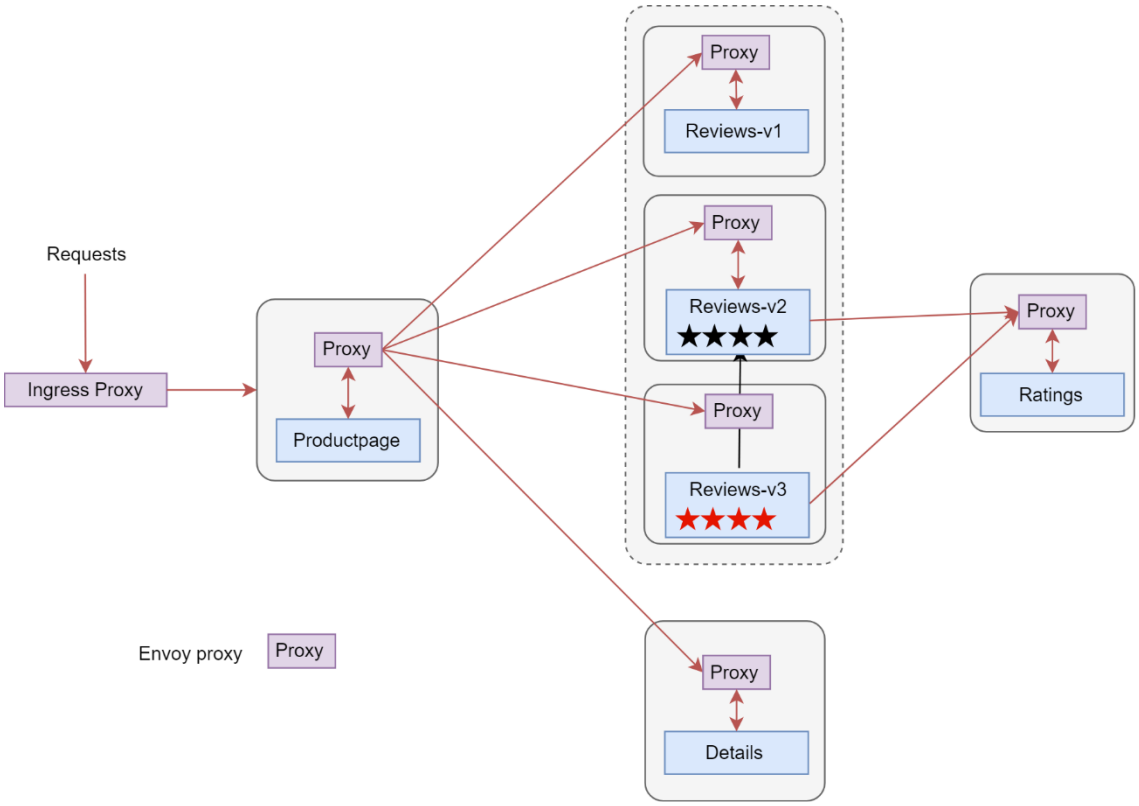

Figure 14. *Bookinfo architecture with sidecars. (From Istio* [4]*)*

In order for Istio to correctly direct the application's traffic through the Istio's load balancer, Istio-ingressgateway, a gateway must be created and applied to Istio to use the load balancer service. It is necessary to do it because the routing rules and other settings applied by the admin will not get used if the requests are not transmitted through the ingress gateway.

After setting up the gateway, different virtual services can be defined that will contain rules for mapping addresses of application services to the targeted service.

In the Bookinfo example, without introducing the gateway, the only way to contact the Productpage is through its service IP address, in the figure it can be seen that the request is not coming from Istio-ingressgateway and therefore the rules applied to the ingress won't get applied to the application. After setting the Bookinfo gateway and the virtual service, we can query the ingress with the targeted application URL and then the requests will be routed from ingress point.

```
apiVersion: networking.Istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
```

```
spec:
  servers:
  - hosts:
      - "*"
  - port:
      number: 80
      name: http
      protocol: HTTP
```

Program 3. *Gateway yaml script.*

In the code above, it is defined so that the gateway is now listening to accept requests for matching from all the hosts.

The virtual service script which introduces the URL matching for different applications deployed will look like the following code, that declares which gateway to use and dispatch the requests related to the URI to the destination host.

```
apiVersion: networking.Istio.io/v1alpha3
kind: VirtualService
spec:
  hosts:
  - "*"
  gateways:
  - bookinfo-gateway
  http:
  - match:
    - uri:
        exact: /productpage
    route:
    - destination:
        host: productpage
        port:
          number: 9080
  - match:
    - uri:
        exact: /ping
    route:
    - destination:
        host: simple-webserver
        port:
          number: 8082
```

Program 4. *Virtualservice yaml script code belonging to Bookinfo and simple-webserver example.*

In the figure below which is from the none TLS encrypted Istio setup, since the gateway is not defined, the requests are being sent from browser directly to the Productpage

service's address at http://10.111.37.207:9080/productpage which is shown with unknown. The same applies with the simple web server which is just calling the frontend's address at http://10.98.226.67:8082/ping and that makes the call to the Redis database.

It is worth noting that with the TLS encrypted Istio, the applications are not reachable directly and in order to access them the ingress gateway must be set first to access Istio enabled apps. the reason being that the sidecars are now listening for TLS requests and plain text requests sent to them get discarded.

In Figure. 15 below which is from the none TLS encrypted Istio setup since the gateway is not defined, the requests are being sent from browser directly to the Productpage service's address at http://10.111.37.207:9080/productpage which is shown with unknown. The same applies with the simple web server which is just calling the frontend's address at http://10.98.226.67:8082/ping and that makes the call to the Redis database.

It is worth noting that with the TLS encrypted Istio, the applications are not reachable directly and in order to access them the ingress gateway must be set first to access Istio enabled apps. the reason being that the sidecars are now listening for TLS requests and plain text requests sent to them get discarded.

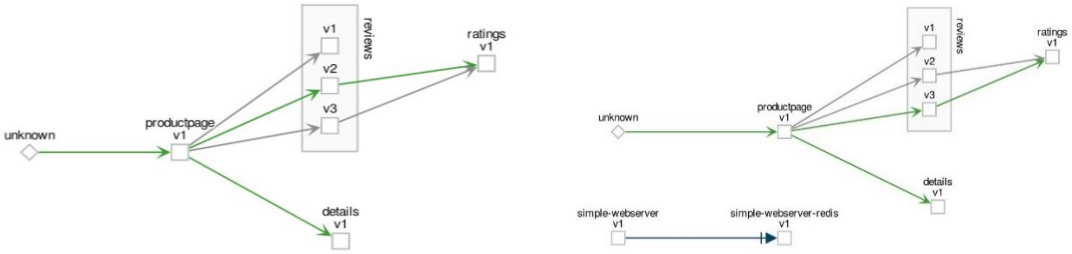

Figure 15. *Network graphs generated by Kiali –Bookinfo and Simple-web-server.*

However, as shown in Figure. 16 below, the traffic had been generated to both Bookinfo and simple webserver. After applying the gateway and virtual services, any call to the ingress's address followed by the URI of the desired application defined in the gateway rules will now flow through the ingress and discards any other invalid requests.

For this particular application of the simple web server, since Redis uses their own RESP/TCP protocol which is also has been drawn with blue arrows, indicating a TCP connection, the telemetry data sent to Jaeger does not get recognized because it only lists and captures the HTTP requests sent by envoys.

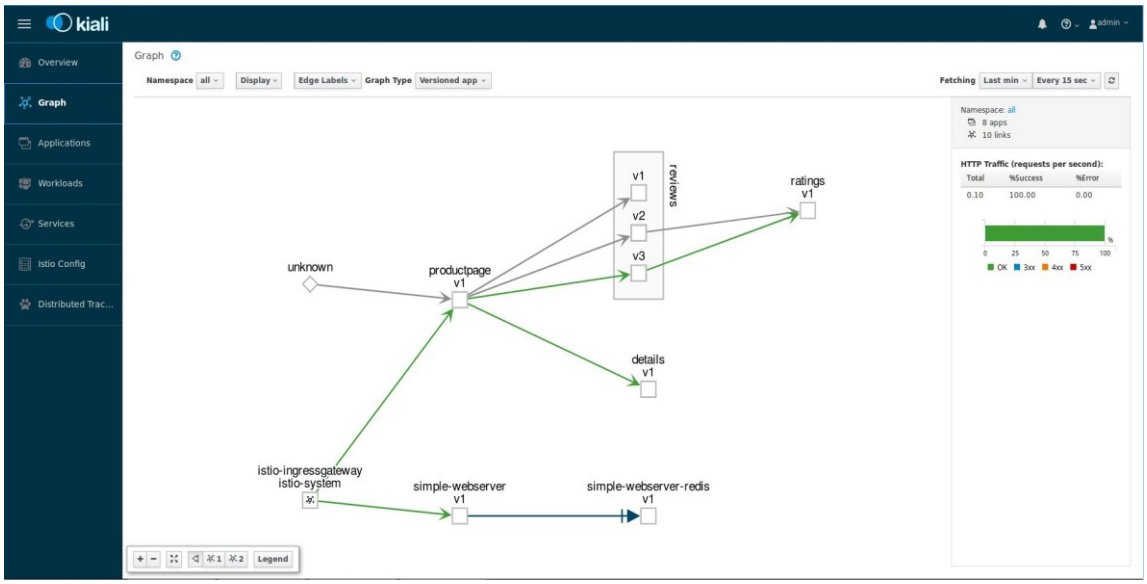

Figure 16. *Network graphs generated by Kiali showing ingress.*

4.3 Latency Measurements

The aim and purpose of this measurement are to be able to say how much extra delay is introduced by the Envoy proxies added to the application. These measurements are important because ultra-low latency required in 5G Core NFs.

Several methods and approaches have been tested to measure the latency of different application, like Bookinfo with Istio service mesh.

One way to monitor the application traffic deployed on Istio is the Jaeger service. In the figure below a detailed trace view of a request to Productpage is shown. It gives a detailed presentation of every hop of the traffic flow and operation which are also known as spans and how long each one takes.

In Figure. 17 all the main remote procedure calls on the tracing is normally illustrated with one span from client side, for example in the case of Details, “productpage details.default.svc.cluster.local:9080/*” and next one the server-side span which is a child process of the previous operation, “details details.default.svc.cluster.local:9080/*”.

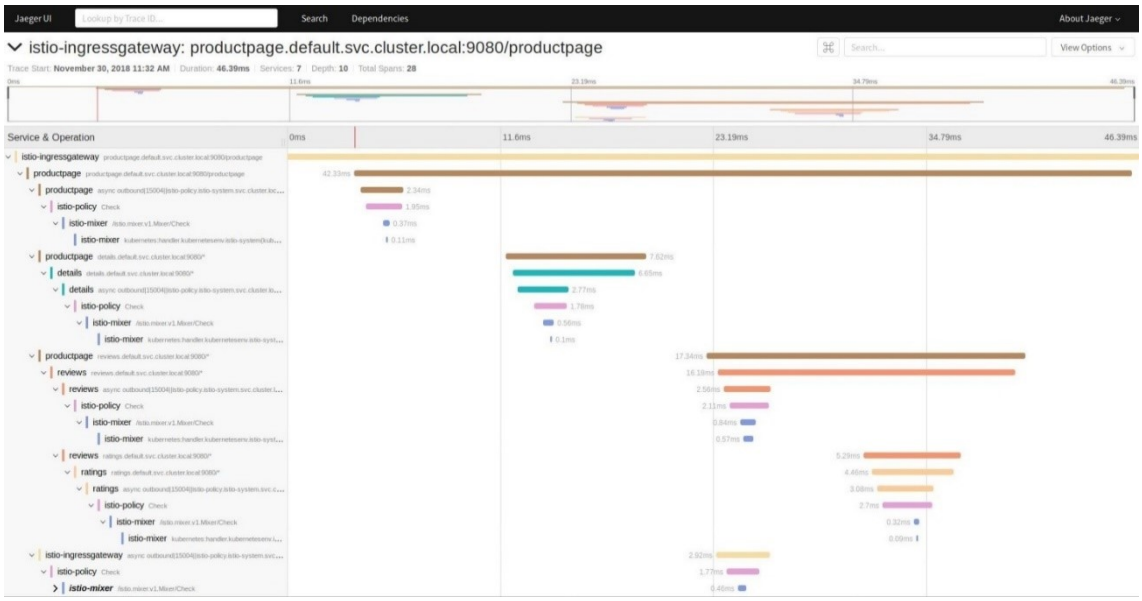

Figure 17. *Istio Observation tracing tools (Jaeger).*

As explained previously, the Rating service is, in fact, dependent on Reviews service to get called, therefore, the call to Reviews spawns two other RPC spans, one for Review itself and other for the Rating service.

For the actual latency measurements three main tools have been used (CURL, WRK and Fortio).

Fortio (originally designed for Istio's performance testing) and WRK are both tools designed for HTTP load generation and report the detailed latency views and statistics.

Another used method for latency measurement is generating the requests by **cURL** command which lets you see many different details of the requests being made including the timing aspects.

To do so, a formatted text file - curl-format.txt - containing the script below was created.

```
            time_namelookup:  %{time_namelookup}\n
               time_connect:  %{time_connect}\n
            time_appconnect:  %{time_appconnect}\n
           time_pretransfer:  %{time_pretransfer}\n
              time_redirect:  %{time_redirect}\n
         time_starttransfer:  %{time_starttransfer}\n
                            ----------\n
                 time_total:  %{time_total}\n\n
```

Program 5. *Curl script used for latency measurement.*

For the purpose of automation and fast load traffic generation, the following command can start to generate the desired number of requests to the address.

```
  for i in {1..300}; do curl -w "@curl-format.txt" -o /dev/null -s http://<Service-IP>/app; done
```

As a result of the faster requests being placed, they don't need to contact Mixer Policy that often resulting in reduced latency.

It must be noted that the envoys have their own cache and can store information about the route and policies within themselves for a limited time that results in fewer spans and so decreased latency. As shown in the figure below, there are two traces presented, one generated as the first request whereas the other one after several sent requests. It clearly indicates that the envoys have a caching mechanism to not contact the Mixer for every request and therefore less overhead.

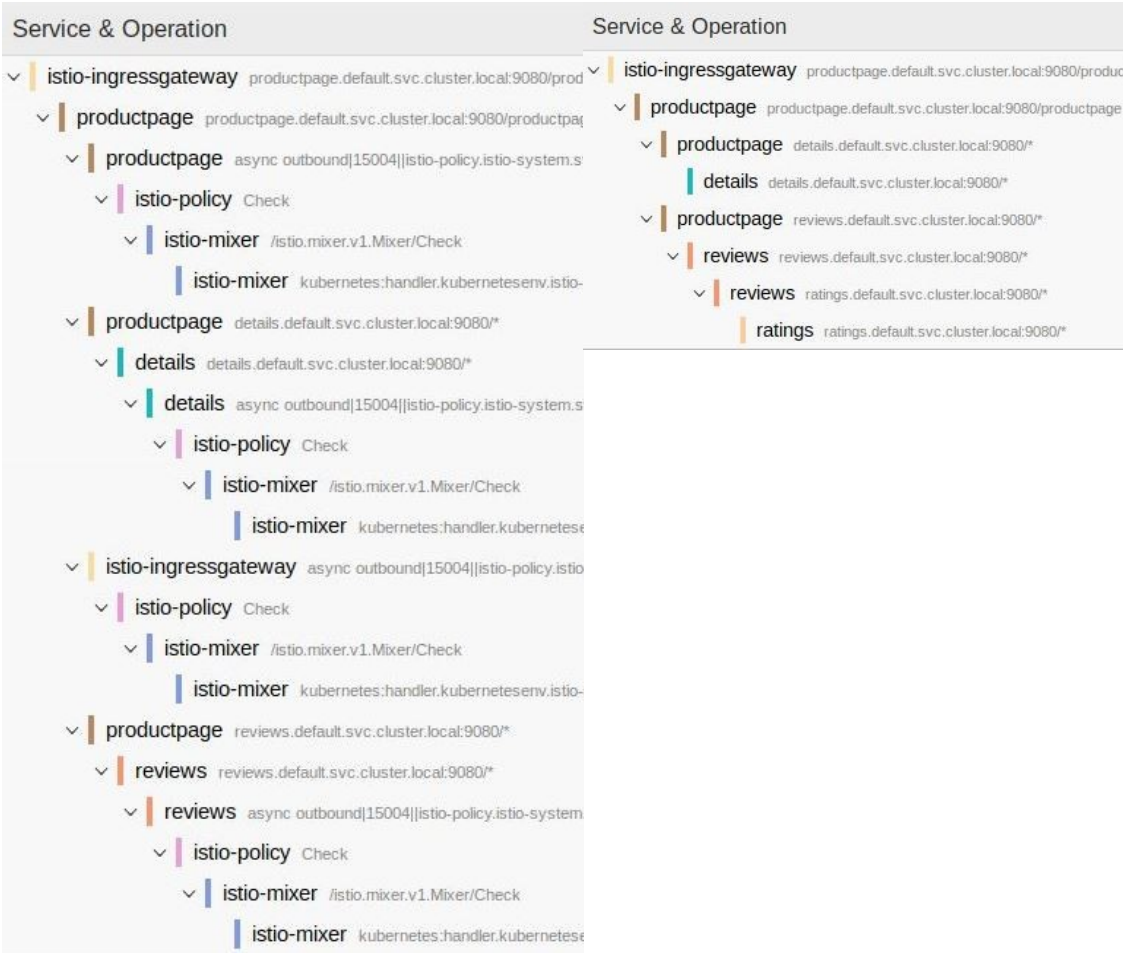

Figure 18. *Jaeger detail request tracing.*

After extensive experiments it was learned and derived that the Bookinfo application itself is not capable of handling high load and it would be bottleneck for the testing, as a result it is not a good example to measure the latencies and the envoys delays, rather it is mainly used to demonstrate the capabilities and different set of services and functions provided by Istio and not for performance testing.

to overcome this problem, the application must not be a bottleneck, therefore, an Nginx webserver was deployed to see how Istio performs when we don't face limitation on the app side. The Nginx app was tested both with and without Istio.

To get even a better test setup, two more Nginx services which were set to proxy mode was deployed before the webserver mode that would only forward the traffic to the next hop and each would get its own sidecar as well. two more instances of the Nginx web-servers were launched as well to get a nice chain of microservices which in practice would also act similar to the 5G Core NFs. This setup provides the possibility to set different routing rules accordingly, i.e. request routing to specific version (instance) of the Nginx web server based on provided headers or cookies. The visual structure of this scheme is seen in figure below.

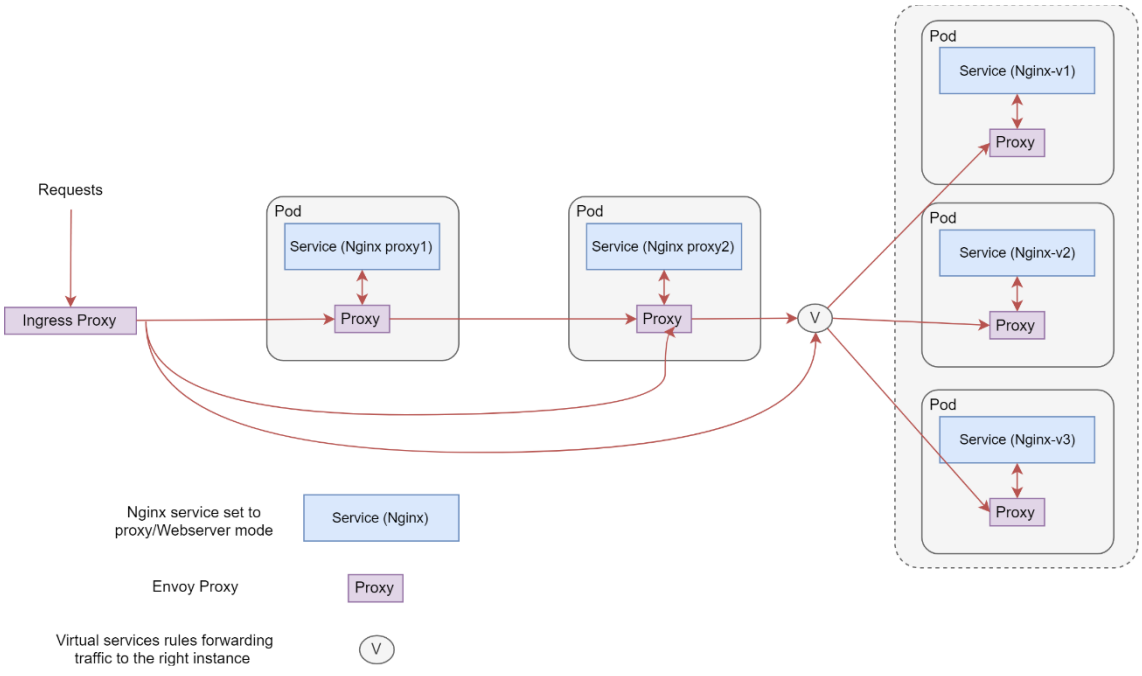

Figure 19. *Nginx application architecture deployed with Istio.*

It is important to mention that the results of the measurements are relative to the test environment and can vary based on the what type of setup has been used, either physical or technology-wise like Docker containers or VMs as Kubernetes nodes. Another important factor that affects the performance and latency is the number of features enabled on Istio and enabling features like tracing and logging can have more impact on the performance testing and results however the Istio profile that was chosen to be tested with is a full-featured one.

However, in this thesis the test scenario is as mentioned above, (Minikube with defined resources). For the sake of getting more consistent and reliable results, the Istio policy check has been disabled which means the Mixer will not be contacted and as a result, the number of spans will be the same for all the requests.

4.4 Performance Analysis

After testing with all the previously mentioned tools and methods, the results on the tables 1 to 3 have been generated showing the latencies introduced in every scenario.

In the first table, the results are from the Nginx sample application but deployed only on Kubernetes regardless of Istio to be able to differentiate with and without the framework. It must also be indicated that all the measurements were done with Istio v1.0.4 at the time.

Table 1. *Kubernetes only setup – No sidecars added.*

NO-Istio	**avg**	**qps**	**total number of reqs in 3 sec**	**Max**	**Min**	**p75**	**p99**
client → Nginx	0.225 ms	999.8	3000	9.81 ms	0.1 ms	0.22 ms	0.88 ms
client → Proxy-2	0.525 ms	999.6	3000	9.08 ms	0.25 ms	0.58 ms	1.48 ms
client → Proxy-1	0.812 ms	999.5	3000	7.71 ms	0.43 ms	0.92 ms	1.88 ms

The second table shows the result from the same application deployed with Istio framework but without enabling the TLS authentication.

Table 2. *Non-TLS Istio setup.*

Non-TLS	**avg**	**qps**	**total number of reqs in 3 sec**	**Max**	**Min**	**p75**	**p99**
Istio-Ingress → Nginx	1.95 ms	502	1511	14.61 ms	0.98 ms	2.23 ms	4.14 ms
Istio-Ingress → Proxy-2	4.10 ms	237	720	20.10 ms	2.41 ms	4.57 ms	8.47 ms
Istio-Ingress → Proxy-1	6.45 ms	153	467	22.92 ms	4.34 ms	6.84 ms	13.6 ms

The third table here shows the results from when the TLS authentication between the sidecars have been enabled which as expected adds slight latency to the total load.

Table 3. *TLS enabled Istio.*

TLS	**avg**	**qps**	**total number of reqs in 3 sec**	**Max**	**Min**	**p75**	**p99**
Istio-Ingress → Nginx	2.29 ms	427	1281	18.85 ms	1.19 ms	2.62 ms	4.21 ms
Istio-Ingress → Proxy-2	4.88 ms	202	608	19.73 ms	2.89 ms	5.47 ms	8.96 ms
Istio-Ingress → Proxy-1	7.86 ms	126	379	17.31 ms	4.96 ms	8.57 ms	11.64 ms

To demonstrate the changes and better representation of the measurements, the figures below were generated by using Fortio tool to benchmark and measure the latencies under the desired load.

Note: More figures and measurements can be found in the appendices section.

The following figure is the results of the targets service communication without sidecars on the left side and when sidecars added to each service component showing in the right

side. it can be understood that one way through a single sidecar adds ~0.5 ms however the absolute numbers will vary according to environment specifics.

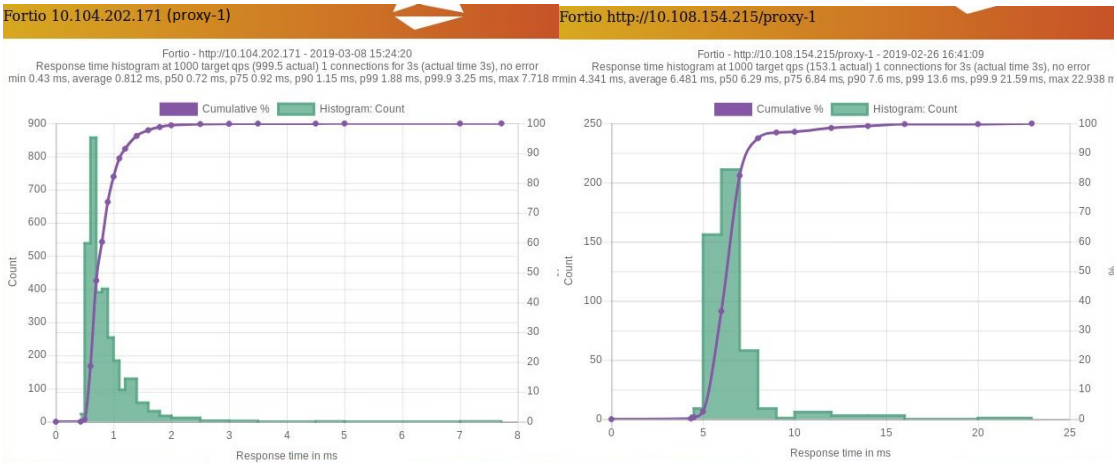

Figure 20. *Fortio generated network load graph comparing with or without sidecars.*

4.5 Istio with Mutual TLS

The Citadel service, which is responsible for the keys and certificate distribution and management, has to be successfully deployed for the TLS to work perfectly.

Istio naturally injects the required keys and certificates to proxy containers of each service in the service mesh regardless of the mutual TLS being enabled or not, where they can be viewed by inspecting inside the docker containers in the path /etc/certs containing the Envoy certificates and private key files along with root certificate file which is, in fact, our Citadel cert to be checked with.

The issue regarding where the keys and certificates are being generated from is addressed by Citadel, it automatically creates a self-signed certificate and key and distributes them to the sidecars upon creation in the cluster and if enabled, it signs their traffic in between.

Another aspect of this setup is that the requests are authenticated only in between the sidecars therefore inside the cluster, however, the traffic from outside the cluster is not, for example, from the browser to the cluster -ingress- (or cURL in terminal).

Whether the service mesh platform will work normally, if only the keys and certificates are taken away to have a none TLS Istio was investigated. In practice there is a configuration fed to the framework and it works seamlessly.

To verify the level of support and robustness of the platform in the context of TLS, the experiment was tried out with first having a non-TLS Istio setup up and running with the applications and enabling the TLS authentication between the sidecars during the process.

To enable mutual TLS authentication in a global manner across all the sidecars in the platform, a mesh policy must be applied throughout the service mesh.

Doing so in practice tells the receiving sides to accept only the authenticated requests, therefore when making a new request from the application services will fail because the client side is still sending their traffic using plain text. To solve the problem the destination rules must be created to make the requests sender services use the mutual TLS to encrypt their traffic. The result of the mentioned method is successful, and the encryption can be applied even during a running platform as well as disabling it by removing the rules and policies.

4.6 The 5G Use Case with Istio

5G emulators are a collection of emulated network functions of 5G technology. An emulated 5G node is a software reference implementation of a certain network functions. In this experimental system UE proving emulated mobile handset functionality and two 5G base stations are deployed. Another network prototype provides implementation of 5G Core SBA and network functions described in section 2.1. The two systems combined provide a complete 5G system deployed for this test setup.

The emulators are constructed in a way to imitate a complete 5G or LTE network and act, serve and produce the same data as if they are running in a real environment for test purposes.

To test the service mesh with mentioned 5G Core components, at first, the required functions need to be extracted and be able to work independently in an isolated environment for some limited functionality and with other parts that are vital for our research such as NRF and AMF.

After getting the functions as independent modules, to introduce them to the Kubernetes like an application, they must be containerized with Docker and turned into images, so they can be deployed on the platform.

The way to build for example the NRF and container with docker is to create an image and creating the containers based on those images. The command below reads the Dockerfile and tags the image with “5G”:

```
docker build -t 5G -f Dockerfile-alpine.
```

In order to reduce the image size, the Alpine Linux distribution which has a small footprint on the containers with around 5Mb has been used.

A Dockerfile script was written and used to create an image on the host machine which installs and contains the necessary packages for the applications and then mounting the source files externally to the launched containers. The reason for this way of deployment is much more flexibility and change the code and configurations on the fly.

However, one issue is the injection and introduction of the certificates, and handling of the configurations. The intended and agreed method is to have them inserted from an external volume rather than write them into the containers by default to be able to create and use new certificate(s) when needed (subject-alt-name modification) and be able to influence the NRF behavior from outside the container on the fly. For this purpose, external volumes must be mounted every time they are brought up.

After successfully building the image and running the NRF container using Docker only, to be able to interact with this application, there are several functions baked into it like Add, Delete and Search the NF profiles for testing purposes. In order to deploy it into Kubernetes, the YMAL script for both the Service and Deployment had to be written.

4.6.1 5G Core Functions as Microservices

After getting some reliable results from the Nginx setup, simulating the main scenario we have a fairly accurate estimation on delay introduced by the envoy proxies in our environment and can be used as the benchmarking layer and comparing the results and be able to identify added latency.

The functions were previously working on VMs, therefore, they first had to be extracted decoupled and made into microservices as standalone functions and from there they were made into docker images and containerized. All the microservices were orchestrated by Kubernetes and by utilizing Istio, the sidecars were added to the pods.

It is worth noting that since the 5G functions use rather different communication protocols like TCP and therefore some points may not be traceable by Jaeger as the reason being that only the HTTP telemetry data is understood by Jaeger. For example, if two microservice like the Simple Webserver application mentioned earlier as it uses TCP between the frontend and the Redis database, we don’t get any tracing capabilities on this application by Jaeger.

ELK server also was setup during the work on another physical server and is ready to receive data logs for analyses and can be used later to get a better insight of what the functions are actually doing since they produce a large number of various logs compare to small portion of it which is visible from the container logs.

The security phase is setup with NRF component and is written in such a way to be able to function over two ports simultaneously. It works on Port A over HTTP/2 over TLS (also defined by 3GPP standards requirements), and on port B which is HTTP/1.1 over TCP (no certificates and keys needed here). However, for better stability even on port A, in case the requester does not support HTTP/2, it switches the request from HTTP/2 to HTTP/1.1 as illustrated in Figure. 21 (still on the port A) and uses the certs for authentication.

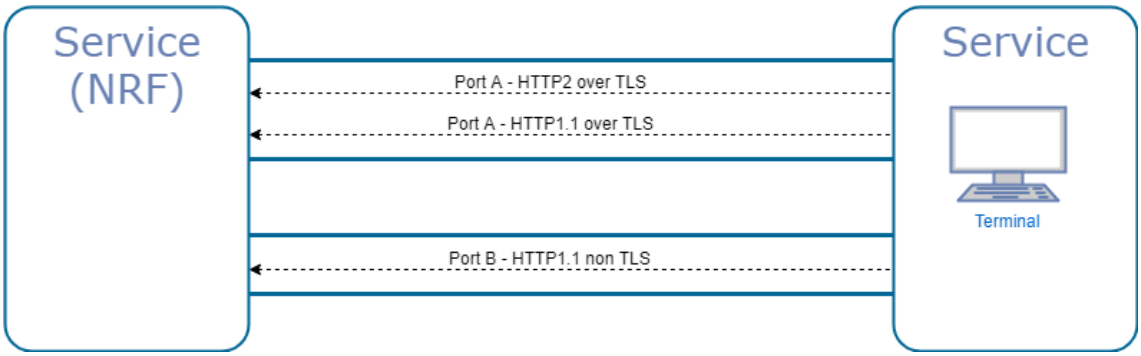

Figure 21. *NRF connectivity and adaptability.*

Istio framework also allows the HTTPS services (applications) to be deployed and perform normally whether the TLS between the sidecars is disabled or enabled. The logic behind how Istio can work with HTTPS services is that the proxy of the HTTPS service which is deployed, changes from operating in Application Layer to Transport Layer regardless of the Istio TLS being enabled or not.

The first thing to be implemented is the creation of the TLS certificate and key by OpenSSL tool to be used by the service to encrypt the traffic. It is notable to add that from the observation during the implementation, the newly generated certificates are automatically injected and added to all the other services' containers by Istio and therefore that is how they can communicate with each other over HTTPS. In simpler way the sidecars still have their own key and cert generated by Citadel and their services have the cert and key which was generated by us.

Next is to create the Kubernetes secret which is in basic terms an object that contains the certs and keys of service and passed to the proper pods.

4.6.2 Connectivity Verification

To verify the connectivity to the NRF service from another service internally is to SSH to another application container and send the requests to NFR.

As a result, another Docker container with very minimal functionality for curl with HTTP/2 support and does nothing else but set to sleep to be deployable into Kubernetes. The application is deployed in the automatic sidecar injection namespace and consequently gets its own sidecar.

Connecting to the running containers by SSH, allows to make the curl request to NRF from that endpoint (terminal) and sending and HTTPS curl with HTTP/2 flag now works perfectly and the request is visible in the logs from the NRF app. Nonetheless, the same request will not go through when generated from the proxy containers of the applications as expected due to the security and how the TLS behaves because the proxies don't have the certs which were manually created and were passed to the service's container.

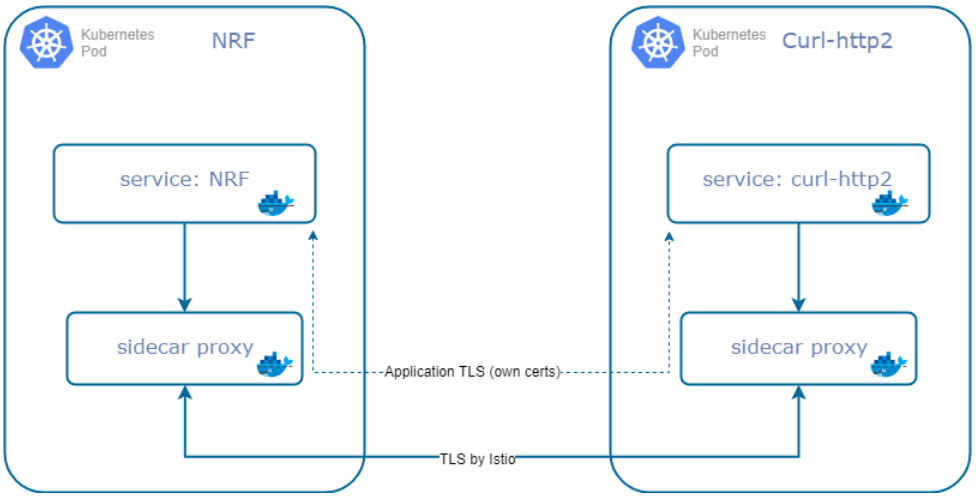

Figure 22. *Connection between NRF and other services.*

Repeating the same request but from another application container without having an HTTP/2 curl still works over the port A and gets authenticated but gets downgraded to HTTP/1.1 as it must.

After running all the tests and checking different issues by running standalone containers to look for any problems, all the Core NFs were deployed on Istio. The figure below shows a visual representation of the mesh network which was generated by Kiali.

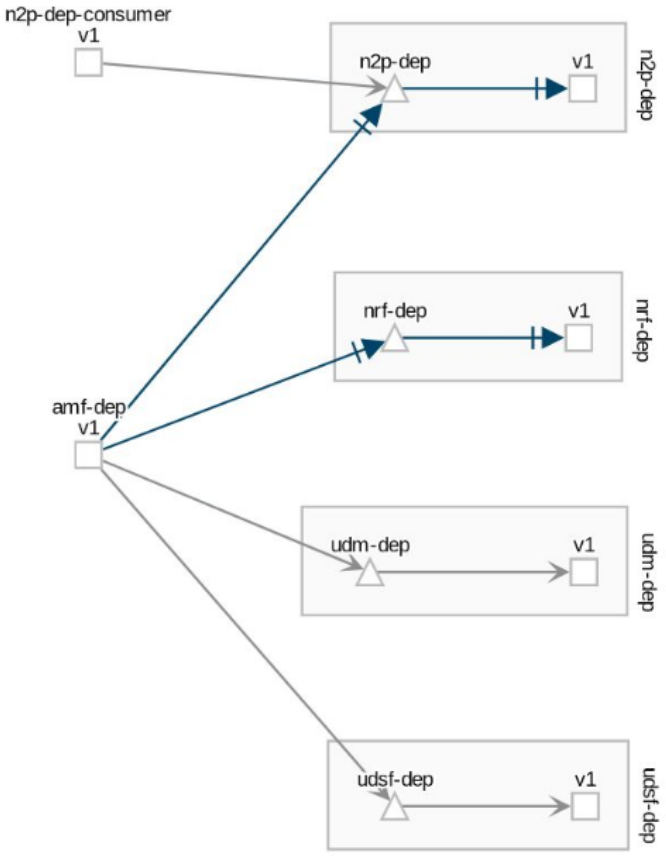

Figure 23. *Network graph generated by Kiali.*

4.7 Multicluster Istio

The purpose of a multicluster service mesh is in situations when there are applications with large sizes and need to be run in a distributed manner on different clusters, for example on different geographical locations. However, the point of running this setup being that all the applications on all clusters being controlled under one or multiple synchronized control planes, and from the 5GC point of view that why this practice is needed, 5G networks are typically such large and business critical deployments, where multicluster deployments would be mandatory.

There are several ways for setting up the multicluster Istio service mesh depending on the underlying platform and infrastructure but the preferred and chosen method was to have a single control plane Istio and that cluster containing the main components be able to access and configure other Istio enabled clusters.

The environment was the first setup by using virtual machines as real clusters on different servers interconnecting them and then the Istio control plane was deployed, the end result was as expected and all the applications and their corresponding sidecars which

were deployed on the secondary cluster were configured by the primary Istio cluster which was clear by following the logs from both ends.

The figure below shows the overall architecture of how a single control plane Istio service mesh works.

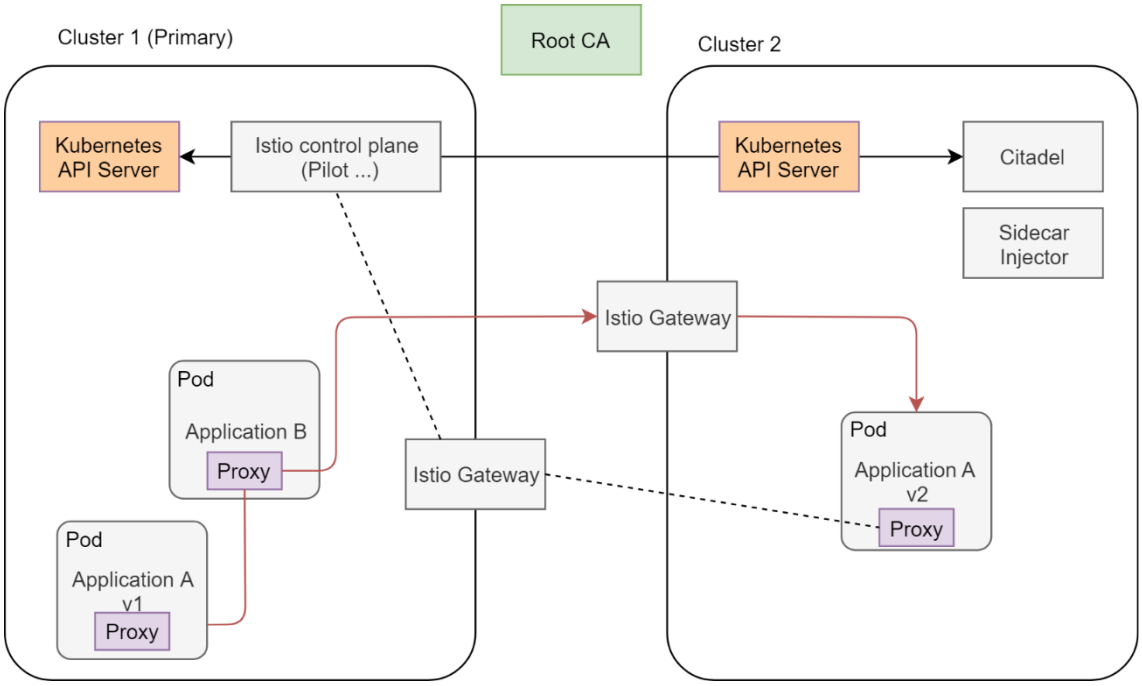

Figure 24. *Single control plane multi cluster based on Istio documentations.*[4]

5. CONCLUSION

In conclusion, there have been several both benefits and drawbacks to using Istio as a service mesh framework as well as some important key learnings for Istio and Kubernetes feasibility for 5GC eSBA solution.

To tell some benefits, Istio by itself is a capable service mesh framework and it could be a usable solution even in terms of 5GC as it is being consequently developed and introduces new functionality needed for 5G rollout and newer evolutions. It has a scalable architecture that can span and support different needs. Load balancing and robustness due to use of proxies and also a very important feature that provides is a high level of observability and tracing on the microservices. There are some important findings worth mentioning like security provided by Istio rather the NFs and the existence of persistent NF names rather i.e. IP addresses that are subjected to change.

However, there are also some downsides to it like, the high degree of dependency on the actual infrastructure (Kubernetes) and complexity of the configuration especially in the case of on private cloud deployment since it is mainly designed to work on public clouds for example.

Despite the work and the results from it, some potential challenges and resolutions have been found in the context of 5G and 3GPP and its feasibility. In 3GPP the overall understanding of the future network deployment unclear for example, if the expectations about the deployment platform cannot be made, the interoperability, etc. requirements might limit Istio for NF internal deployment and SCP as transparent proxies. Service mesh between the NFs might lose the main potential and force the SCP function to 3GPP application-specific. Standardization like (3GPP) specifies the functionalities, which naturally are part of platform (like failure and overload protection etc.). Deeper integration of 3GPP NFs and deployment platforms (Kubernetes etc.) via adapters might help to resolve that.

The 3GPP requirement for the NF E2E security (TLS between service consumer and producer) would make the intermediate load balancers unaware of the header info (preventing intelligent operation). This is a challenge, when Istio deploys automatically the hop-by-hop TLS, which requires further investigation.

In the end the applicability of Istio for SBA in 5GC is something that requires more investigation but from the latency overhead point of view that is introduced by Istio’s sidecars, according the findings on this thesis it is a high potential framework because of small overhead (~0.5 ms per sidecar) but however it is still going to depend a lot on what scale

this framework would be used, for example where there are thousands of microservices, its going to add up to this number which might not be suitable for some latency critical applications.

REFERENCES

[1] Nokia, “Nokia AirGile cloud-native core: generate new economic value on any access including 5G,” 2019.

[2] Nokia, “Why does 5G need a service mesh?”

[3] G. Miranda, “Which Service Mesh Should I Use,” 2018. [Online]. Available: https://thenewstack.io/which-service-mesh-should-i-use/. [Accessed: 22-Jul-2019].

[4] Istio, “What is Istio,” 2019. [Online]. Available: https://istio.io/. [Accessed: 03-Dec-2018].

[5] H. Abie, “Adaptive security and trust management for autonomic message-oriented middleware,” in *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, 2009, pp. 810–817.

[6] D. A. Chappell, *Enterprise service bus*. O’Reilly, 2004.

[7] 3GPP TS 23.501, “3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; System Architecture for the 5G System; Stage 2 (Release 15),” 2017.

[8] 3GPP TR 29.891, “3rd Generation Partnership Project; Technical Specification Group Core Network and Terminals; 5G System – Phase 1; CT WG4 Aspects (Release 15),” 2017.

[9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture,” *IEEE Software*. 2016.

[10] K. Jackson Higgins, “Forrester Pushes ‘Zero Trust’ Model For Security,” 2010. [Online]. Available: https://www.darkreading.com/attacks-breaches/forrester-pushes-zero-trust-model-for-security/d/d-id/1134373. [Accessed: 20-Nov-2018].

[11] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Migrating to Cloud-Native architectures using microservices: An experience report,” in *Communications in Computer and Information Science*, 2016.

[12] J. Wilke, “5G Network Architecture and FMC,” 2017.

[13] E. Dahlman *et al.*, “5G wireless access: Requirements and realization,” *IEEE Commun. Mag.*, pp. 42–47, 2014.

[14] M. Rahnema and M. Dryjanski, *From LTE to LTE-Advanced Pro and 5G*. 2017.

[15] J. Kim, D. Kim, and S. Choi, “3GPP SA2 architecture and functions for 5G mobile communication system,” *ICT Express*. pp. 1–8, 2017.

[16] TS 29.502, “3rd Generation Partnership Project; Technical Specification Group Core Network and Terminals; 5G System; Session Management Services; Stage 3 (Release 15),” 2018.

[17] A. Kaloxylos, “ENABLING 5G VERTICALS AND SERVICES THROUGH NETWORK SOFTWARIZATION AND SLICING A Survey and an Analysis of Network Slicing in 5G Networks,” *IEEE Commun. Stand. Mag.*, pp. 60–65, 2018.

[18] 3GPP TS 29.510, “3rd Generation Partnership Project; Technical Specification Group Core Network and Terminals; 5G System; Network Function Repository Services; Stage 3 (Release 15),” 2018.

[19] M. Agiwal, A. Roy, and N. Saxena, “Next generation 5G wireless networks: A comprehensive survey,” *IEEE Communications Surveys and Tutorials*. pp. 1617–1655, 2016.

[20] S. Singh and P. Singh, “Key Concepts and Network Architecture for 5G Mobile Technology,” 2012.

[21] A. Gupta and R. K. Jha, “A Survey of 5G Network: Architecture and Emerging Technologies,” *IEEE Access*. pp. 1206–1232, 2015.

[22] W. H. Chin, Z. Fan, and R. Haines, “Emerging technologies and research challenges for 5G wireless networks,” *IEEE Wirel. Commun.*, pp. 106–112, 2014.

[23] G. Brown, “Service-Based Architecture for 5G Core Networks,” *A Heavy Read. white Pap. Prod. Huawei Technol. Co. Ltd. Online verf{\"u}gbar unter https//www. huawei. com/en/press-events/news/2017/11/HeavyReading-WhitePaper-5G-Core-Network, letzter Zugriff am*, vol. 1, p. 2088, 2017.
[24] J. Lee *et al.*, “LTE-advanced in 3GPP Rel -13/14: An evolution toward 5G,” *IEEE Commun. Mag.*, pp. 36–42, 2016.
[25] K. Haneda *et al.*, “5G 3GPP-like channel models for outdoor urban microcellular and macrocellular environments,” in *IEEE Vehicular Technology Conference*, 2016.
[26] 3GPP TS 23.501, “3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; System Architecture for the 5G System (5GS); Stage 2 (Release 16),” 2019.
[27] ETSI TS 129 213, “TS 129 213 - V10.2.0 - Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; Policy and charging control signalling flows and Quality of Service (QoS) parameter mapping (3GPP TS 29.213 version 10.2.,” 2011.
[28] Diametriq, “Diameter Routing Agent,” 2019. [Online]. Available: http://diametriq.com/diametriq-products/diameter-routing-engine-diametriqs-diameter-routing-agent/. [Accessed: 20-Feb-2019].
[29] C. Richardson and F. Smith, “Microservices from Design to Deployment. NGINX,” *Nginx Inc*, 2016.
[30] F. Smith and O. Garrett, “What Is a Service Mesh,” 2018. [Online]. Available: https://www.nginx.com/blog/what-is-a-service-mesh/. [Accessed: 20-Jan-2019].
[31] W. Morgan, “What’s a service mesh And why do I need one,” 2017. [Online]. Available: https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/. [Accessed: 20-Nov-2018].
[32] M. HASHIMOTO, “HashiCorp Consul 1.2: Service Mesh,” 2018. [Online]. Available: https://www.hashicorp.com/blog/consul-1-2-service-mesh. [Accessed: 30-Nov-2018].
[33] E. Casalicchio, “Autonomic Orchestration of Containers: Problem Definition and Research Challenges,” 2010.
[34] Docker, “What is a Container,” 2019. [Online]. Available: https://www.docker.com. [Accessed: 02-Nov-2018].
[35] Splunk, “WHITE PAPER: THE ESSENTIAL GUIDE TO CONTAINER MONITORING.”
[36] Kubernetes, “Production-Grade Container Orchestration,” 2019. [Online]. Available: https://kubernetes.io. [Accessed: 02-Nov-2018].
[37] K. Hightower, B. Burns, and J. Beda, *Kubernetes : up and running: dive into the future of infrastructure*. .
[38] T. Allclair, “Kubernetes v1.12: Introducing RuntimeClass,” 2018. [Online]. Available: https://kubernetes.io/blog/2018/10/10/kubernetes-v1.12-introducing-runtimeclass/. [Accessed: 15-Dec-2018].
[39] J. Langemak, “Kubernetes 101 – External access into the cluster,” 2015. [Online]. Available: http://www.dasblinkenlichten.com/kubernetes-101-external-access-into-the-cluster/. [Accessed: 15-Feb-2018].
[40] Helm, “The Package manager for Kubernetes,” 2019. [Online]. Available: https://helm.sh/. [Accessed: 20-Feb-2019].
[41] Elasticsearch, “What is the ELK Stack,” 2019. [Online]. Available: https://www.elastic.co/what-is/elk-stack. [Accessed: 03-Oct-2018].
[42] D. Berman, “Zipkin vs Jaeger: Getting Started With Tracing,” 2018. [Online]. Available: https://logz.io/blog/zipkin-vs-jaeger/. [Accessed: 17-Jan-2019].
[43] CNCF, “Jaeger: open source, end-to-end distributed tracing,” 2019. [Online]. Available: https://www.jaegertracing.io/. [Accessed: 17-Dec-2018].
[44] Prometheus, “Prometheus,” 2019. [Online]. Available: https://prometheus.io/. [Accessed: 17-Jan-2019].

[45] Grafana Labs, “Grafana App for Kubernetes,” 2019. [Online]. Available: https://grafana.com/grafana/plugins/grafana-kubernetes-app. [Accessed: 17-Jan-2019].
[46] Kiali, “Service mesh observability and configuration,” 2019. [Online]. Available: https://www.kiali.io/. [Accessed: 04-Feb-2019].
[47] S. Acreman, “Service Mesh,” 2019. [Online]. Available: https://kubedex.com/istio-vs-linkerd-vs-linkerd2-vs-consul/. [Accessed: 30-Jan-2019].
[48] Linkerd, “What is Linkerd,” 2019. [Online]. Available: https://linkerd.io. [Accessed: 30-Jan-2019].
[49] O. Zimmermann, “Microservices tenets: Agile approach to service development and deployment,” *Comput. Sci. - Res. Dev.*, pp. 301–310, 2017.

APPENDIX A: FORTIO MEASUREMENTS GRAPHS

The figures below are Fortio generated network load graph in different scenarios

Requests sent through the ingress:

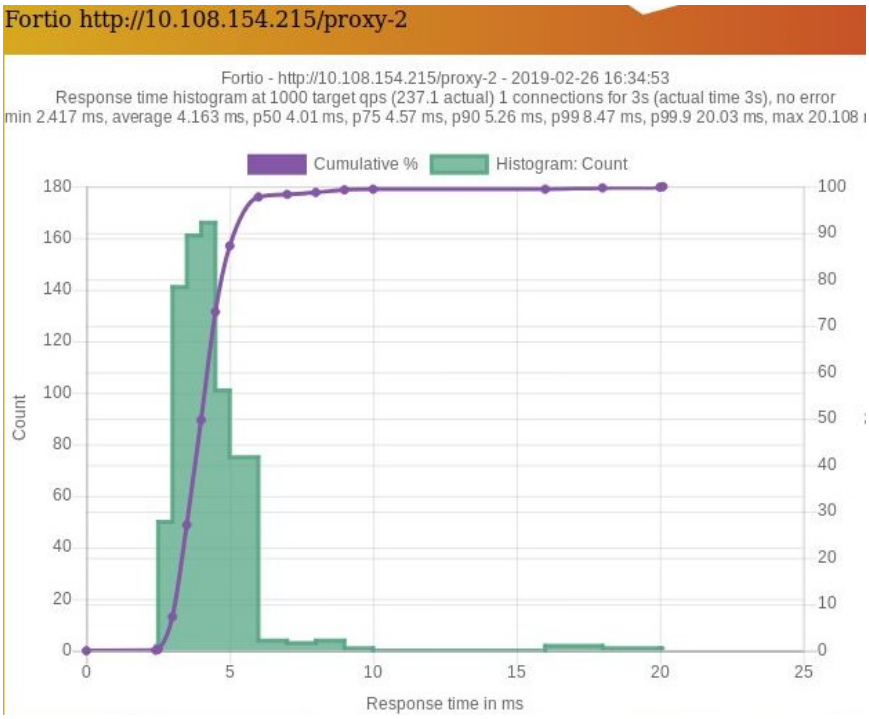

Figure 1. *Sending requests from Ingress to "Nginx proxy 2".*

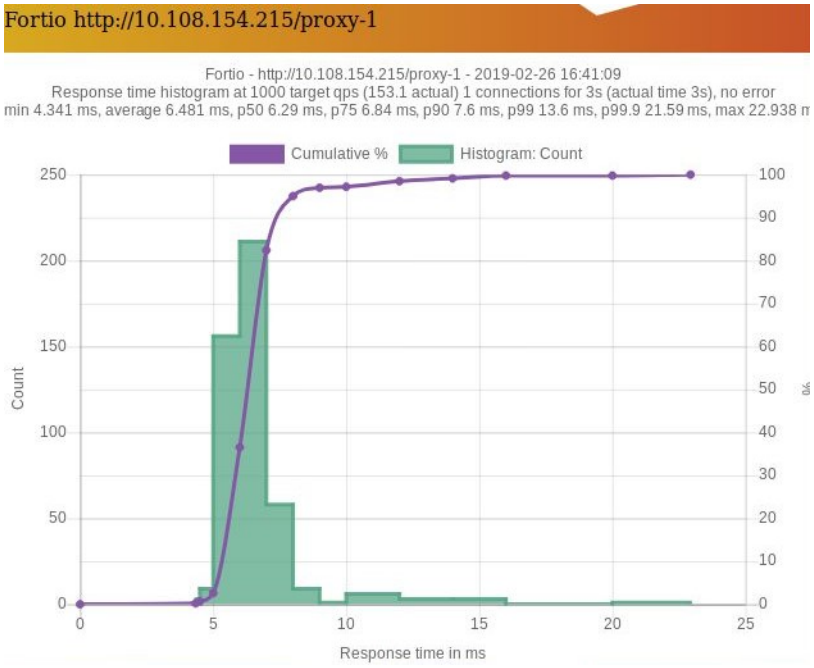

Figure 2. *Sending requests from Ingress to "Nginx proxy 1".*

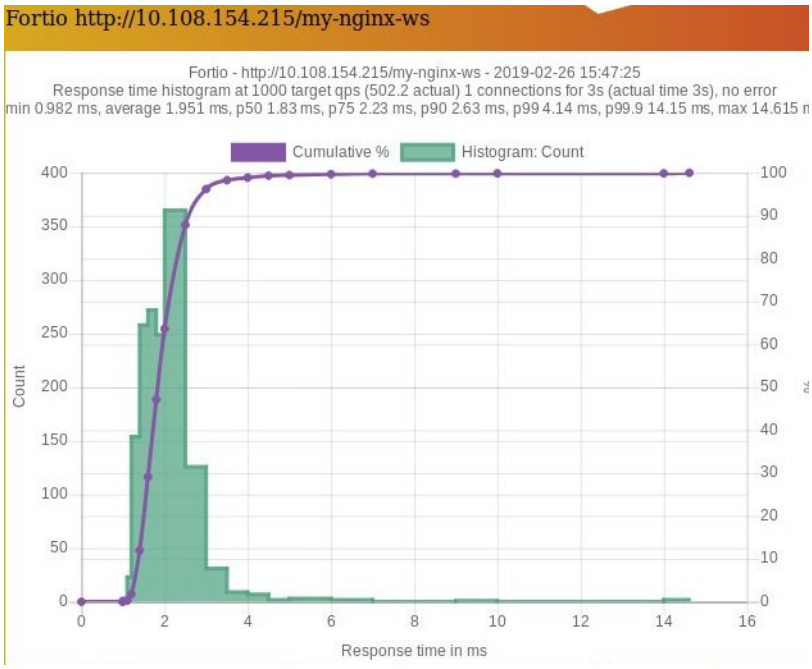

Figure 3. *Sending requests from Ingress to "Nginx-v1" (1 hop).*

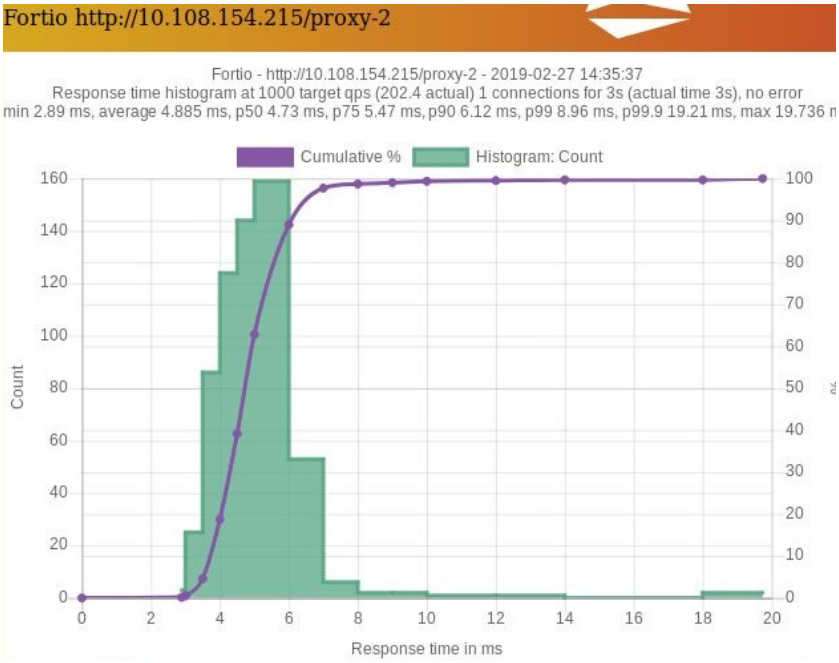

Figure 4. *Sending requests from Ingress to "Nginx proxy 2" with TLS enabled.*

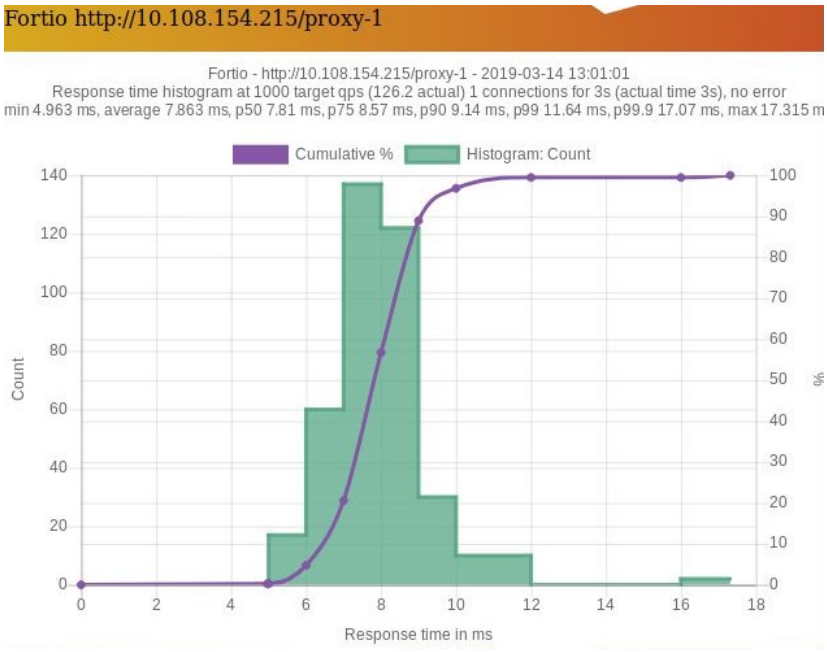

Figure 5. *Sending requests from Ingress to "Nginx proxy 1" with TLS enabled.*

Figure 6. *Sending requests from Ingress to "Nginx-v1" (1 hop) with TLS enabled.*

Requests sent directly to service (without ingress):

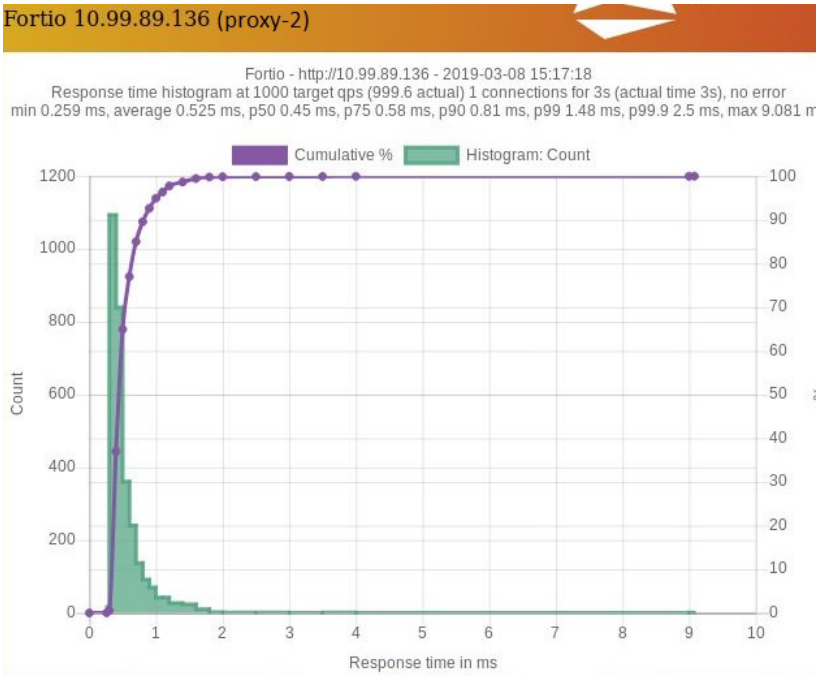

Figure 7. *Sending requests to "Nginx proxy 2" without Istio (only Kubernetes).*

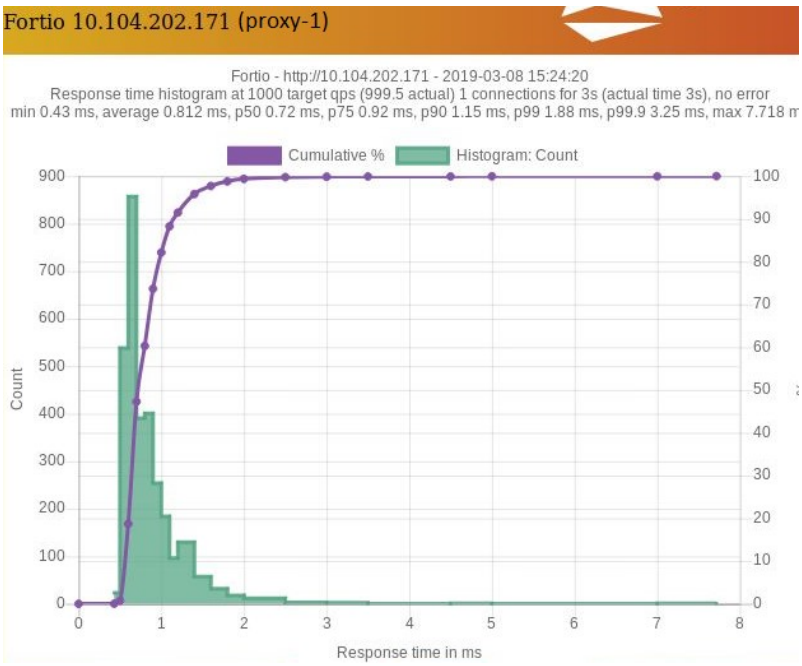

Figure 8. *Sending requests to "Nginx proxy 1" without Istio (only Kubernetes).*

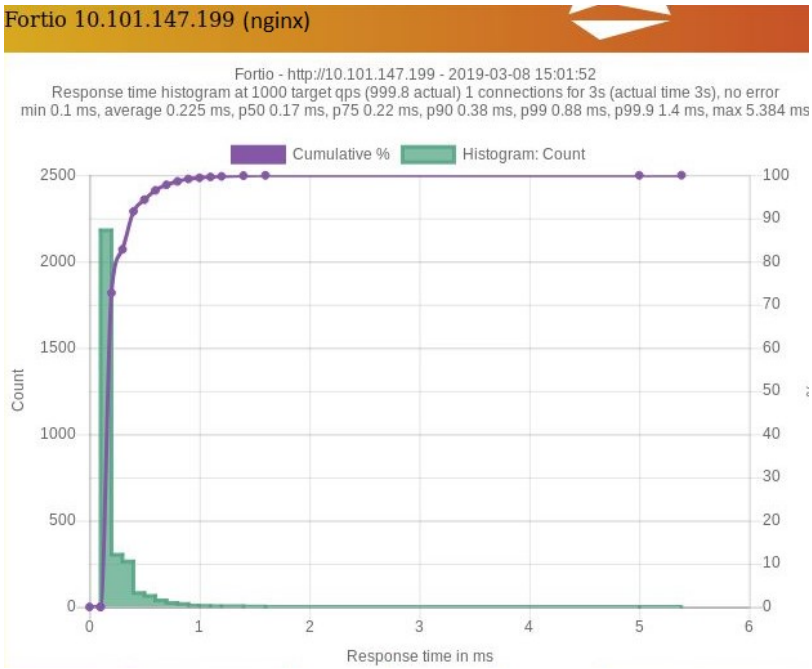

Figure 9. *Sending requests to "Nginx-v1" without Istio (only Kubernetes).*